# 1

# What can't be ignored

In this book we will systematically use elementary mathematical concepts which the reader should know already, yet he or she might not recall them immediately.

We will therefore use this chapter to refresh them and we will condense notions which are typical of courses in Calculus, Linear Algebra and Geometry, yet rephrasing them in a way that is suitable for use in Scientific Computing. At the same time we will introduce new concepts which pertain to the field of Scientific Computing and we will begin to explore their meaning and usefulness with the help of MATLAB (MATrix LABoratory), an integrated environment for programming and visualization. We shall also use GNU Octave (in short, Octave), an interpreter for a high-level language mostly compatible with MATLAB which is distributed under the terms of the GNU GPL free-software license and which reproduces a large part of the numerical facilities of MATLAB.

In Section 1.1 we will give a quick introduction to MATLAB and Octave, while we will present the elements of programming in Section 1.7. However, we refer the interested readers to the manuals [HH05, Pal08] for a description of the MATLAB language and to the manual [EBH08] for a description of Octave.

## 1.1 The MATLAB and Octave environments

MATLAB and Octave are integrated environments for Scientific Computing and visualization. They are written mostly in C and C++ languages.

MATLAB is distributed by The MathWorks (see the website `www.mathworks.com`). The name stands for *MATrix LABoratory* since originally it was developed for matrix computation.

Octave, also known as GNU Octave (see the website `www.octave.org`), is a freely redistributable software. It can be redistributed and/or

modified under the terms of the GNU General Public License (GPL) as published by the Free Software Foundation.

There are differences between MATLAB and Octave environments, languages and toolboxes (i.e. a collection of special-purpose MATLAB functions). However, there is a level of compatibility that allows us to write most programs of this book and run them seamlessly both in MATLAB and Octave. When this is not possible, either because some commands are spelt differently, or because they operate in a different way, or merely because they are just not implemented, a note will be written at the end of each section to provide an explanation and indicate what could be done.

Through the book, we shall often make use of the expression "MATLAB command": in this case, MATLAB should be understood as the *language* which is the common subset of both programs MATLAB and Octave.

Just as MATLAB has its toolboxes, Octave has a richful set of functions available through a project called Octave-forge (see the website `octave.sourceforge.net`). This function repository grows steadily in many different areas. Some functions we use in this book don't belong to the Octave core, nevertheless they can be downloaded by the website `octave.sourceforge.net`.

`>>` `octave:1>` Once installed, the execution of MATLAB or Octave yield the access to a working environment characterized by the *prompt* `>>` or `octave:1>`, respectively. For instance, when executing MATLAB on our personal computer, the following message is generated:

```
                    < M A T L A B (R) >
          Copyright 1984-2009 The MathWorks, Inc.
       Version 7.9.0.529 (R2009b) 64-bit (glnxa64)
                      August 12, 2009


  To get started, type one of these: helpwin, helpdesk, or demo.
  For product information, visit www.mathworks.com.
  >>
```

When executing Octave on our personal computer we read the following text:

```
GNU Octave, version 3.2.3
Copyright (C) 2009 John W. Eaton and others.
This is free software; see the source code for copying
conditions. There is ABSOLUTELY NO WARRANTY; not even
for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
For details, type 'warranty'.
Octave was configured for "x86_64-unknown-linux-gnu".
Additional information about Octave is available at
http://www.octave.org.
```

```
Please contribute if you find this software useful.
For more information, visit
http://www.octave.org/help-wanted.html
Report bugs to <bug@octave.org> (but first, please read
http://www.octave.org/bugs.html to learn how to write a
helpful report).

For information about changes from previous versions,
type 'news'.

octave:1>
```

In this chapter we will use the *prompt* `>>`, however, from Chapter 2 on the *prompt* will be always neglected in order to simplify notations.

## 1.2 Real numbers

While the set $\mathbb{R}$ of real numbers is known to everyone, the way in which computers treat them is perhaps less well known. On one hand, since machines have limited resources, only a subset $\mathbb{F}$ of finite dimension of $\mathbb{R}$ can be represented. The numbers in this subset are called *floating-point numbers*. On the other hand, as we shall see in Section 1.2.2, $\mathbb{F}$ is characterized by properties that are different from those of $\mathbb{R}$. The reason is that any real number $x$ is in principle truncated by the machine, giving rise to a new number (called the *floating-point number*), denoted by $fl(x)$, which does not necessarily coincide with the original number $x$.

### 1.2.1 How we represent them

To become acquainted with the differences between $\mathbb{R}$ and $\mathbb{F}$, let us make a few experiments which illustrate the way that a computer deals with real numbers. Note that whether we use MATLAB or Octave rather than another language is just a matter of convenience. The results of our calculation, indeed, depend primarily on the manner in which the computer works, and only to a lesser degree on the programming language. Let us consider the rational number $x = 1/7$, whose decimal representation is $0.\overline{142857}$. This is an infinite representation, since the number of decimal digits is infinite. To get its computer representation, let us introduce after the *prompt* the ratio `1/7` and obtain

```
>> 1/7
ans =
    0.1429
```

which is a number with only four decimal digits, the last being different from the fourth digit of the original number.

Should we now consider 1/3 we would find 0.3333, so the fourth decimal digit would now be exact. This behavior is due to the fact that real numbers are *rounded* on the computer. This means, first of all, that only an a priori fixed number of decimal digits are returned, and moreover the last decimal digit which appears is increased by unity whenever the first disregarded decimal digit is greater than or equal to 5.

The first remark to make is that using only four decimal digits to represent real numbers is questionable. Indeed, the internal representation of the number is made of as many as 16 decimal digits, and what we have seen is simply one of several possible MATLAB output formats. The same number can take different expressions depending upon the specific format declaration that is made. For instance, for the number 1/7, some possible output *formats* are avalibale in MATLAB:

`format`

```
format short   yields 0.1429,
format short e   "    1.4286e − 01,
format short g   "    0.14286,
format long      "    0.142857142857143,
format long e    "    1.428571428571428e − 01,
format long g    "    0.142857142857143.
```

The same formats are available in Octave, but the yielded results do not necessarily coincide with those of MATLAB:

```
format short   yields 0.14286,
format short e   "    1.4286e − 01,
format short g   "    0.14286,
format long      "    0.142857142857143,
format long e    "    1.42857142857143 − 01,
format long g    "    0.142857142857143.
```

Obviously, these differences, even if slight, will imply possible different results in the treatment of our examples.

Some of these formats are more coherent than others with the internal computer representation. As a matter of fact, in general a computer stores a real number in the following way

$$x = (-1)^s \cdot (0.a_1 a_2 \dots a_t) \cdot \beta^e = (-1)^s \cdot m \cdot \beta^{e-t}, \quad a_1 \neq 0 \tag{1.1}$$

where $s$ is either 0 or 1, $\beta$ (a positive integer larger than or equal to 2) is the *basis* adopted by the specific computer at hand, $m$ is an integer called the *mantissa* whose length $t$ is the maximum number of digits $a_i$ (with $0 \leq a_i \leq \beta - 1$) that are stored, and $e$ is an integral number called the *exponent*. The format `long e` is the one which most resembles this

representation, and `e` stands for exponent; its digits, preceded by the sign, are reported to the right of the character `e`. The numbers whose form is given in (1.1) are called floating-point numbers, since the position of the decimal point is not fixed. The digits $a_1 a_2 \dots a_p$ (with $p \leq t$) are often called the $p$ first significant digits of $x$.

The condition $a_1 \neq 0$ ensures that a number cannot have multiple representations. For instance, without this restriction the number $1/10$ could be represented (in the decimal basis) as $0.1 \cdot 10^0$, but also as $0.01 \cdot 10^1$, etc..

The set $\mathbb{F}$ is therefore fully characterized by the basis $\beta$, the number of significant digits $t$ and the range $(L, U)$ (with $L < 0$ and $U > 0$) of variation of the index $e$. Thus it is denoted as $\mathbb{F}(\beta, t, L, U)$. For instance, in MATLAB we have $\mathbb{F} = \mathbb{F}(2, 53, -1021, 1024)$ (indeed, 53 significant digits in basis 2 correspond to the 15 significant digits that are shown by MATLAB in basis 10 with the `format long`).

Fortunately, the *roundoff error* that is inevitably generated whenever a real number $x \neq 0$ is replaced by its representative $fl(x)$ in $\mathbb{F}$, is small, since

$$\boxed{\frac{|x - fl(x)|}{|x|} \leq \frac{1}{2}\epsilon_M} \tag{1.2}$$

where $\epsilon_M = \beta^{1-t}$ provides the distance between 1 and its closest floating-point number greater than 1. Note that $\epsilon_M$ depends on $\beta$ and $t$. For instance, in MATLAB $\epsilon_M$ can be obtained through the command `eps`, and we obtain $\epsilon_M = 2^{-52} \simeq 2.22 \cdot 10^{-16}$. Let us point out that in (1.2) we estimate the *relative error* on $x$, which is undoubtedly more meaningful than the *absolute error* $|x - fl(x)|$. As a matter of fact, the latter doesn't account for the order of magnitude of $x$ whereas the former does.

`eps`

The number $u = \frac{1}{2}\epsilon_M$ is the maximum relative error that the computer can make while representing a real number by finite arithmetic. For this reason, it is sometimes named *roundoff unity*.

Number 0 does not belong to $\mathbb{F}$, as in that case we would have $a_1 = 0$ in (1.1): it is therefore handled separately. Moreover, $L$ and $U$ being finite, one cannot represent numbers whose absolute value is either arbitrarily large or arbitrarily small. Precisely, the smallest and the largest positive real numbers of $\mathbb{F}$ are given respectively by

$$x_{min} = \beta^{L-1}, \; x_{max} = \beta^U (1 - \beta^{-t}).$$

In MATLAB these values can be obtained through the commands `realmin` and `realmax`, yielding

`realmin`
`realmax`

$$\begin{aligned} x_{min} &= 2.225073858507201 \cdot 10^{-308}, \\ x_{max} &= 1.797693134862316 \cdot 10^{+308}. \end{aligned}$$

A positive number smaller than $x_{min}$ produces a message of underflow and is treated either as 0 or in a special way (see, e.g., [QSS07], Chapter 2). A positive number greater than $x_{max}$ yields instead a message of overflow and is stored in the variable `Inf` (which is the computer representation of $+\infty$).

`Inf`

The elements in $\mathbb{F}$ are more dense near $x_{min}$, and less dense while approaching $x_{max}$. As a matter of fact, the number in $\mathbb{F}$ nearest to $x_{max}$ (to its left) and the one nearest to $x_{min}$ (to its right) are, respectively

$$x_{max}^{-} = 1.797693134862315 \cdot 10^{+308},$$
$$x_{min}^{+} = 2.225073858507202 \cdot 10^{-308}.$$

Thus $x_{min}^{+} - x_{min} \simeq 10^{-323}$, while $x_{max} - x_{max}^{-} \simeq 10^{292}$ (!). However, the relative distance is small in both cases, as we can infer from (1.2).

### 1.2.2 How we operate with floating-point numbers

Since $\mathbb{F}$ is a proper subset of $\mathbb{R}$, elementary algebraic operations on floating-point numbers do not enjoy all the properties of analogous operations on $\mathbb{R}$. Precisely, commutativity still holds for addition (that is $fl(x+y) = fl(y+x)$) as well as for multiplication ($fl(xy) = fl(yx)$), but other properties such as associativity and distributivity are violated. Moreover, 0 is no longer unique. Indeed, let us assign the variable `a` the value 1, and execute the following instructions:

```
>> a = 1; b=1; while a+b ~= a; b=b/2; end
```

The variable `b` is halved at every step as long as the sum of `a` and `b` remains different (`~=`) from `a`. Should we operate on real numbers, this program would never end, whereas in our case it ends after a finite number of steps and returns the following value for `b`: `1.1102e-16`$= \epsilon_M/2$. There exists therefore at least one number `b` different from 0 such that `a+b=a`. This is possible since $\mathbb{F}$ is made up of isolated numbers; when adding two numbers `a` and `b` with `b<a` and `b` less than $\epsilon_M$, we always obtain that `a+b` is equal to `a`. The MATLAB number `a+eps(a)` is the smallest number in $\mathbb{F}$ larger than `a`. Thus the sum `a+b` will return `a` for all `b < eps(a)`.

Associativity is violated whenever a situation of overflow or underflow occurs. Take for instance `a=1.0e+308`, `b=1.1e+308` and `c=-1.001e+308`, and carry out the sum in two different ways. We find that

$$\texttt{a} + (\texttt{b} + \texttt{c}) = \texttt{1.0990e} + \texttt{308}, \; (\texttt{a} + \texttt{b}) + \texttt{c} = \texttt{Inf}.$$

This is a particular instance of what occurs when one adds two numbers with opposite sign but similar absolute value. In this case the result may be quite inexact and the situation is referred to as *loss*, or *cancellation, of significant digits*. For instance, let us compute $((1+x)-1)/x$

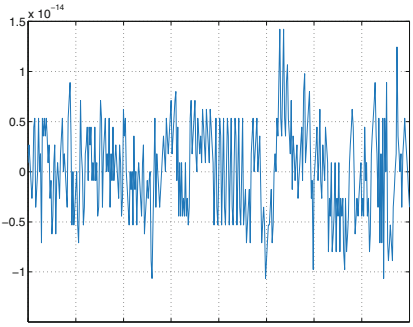


**Fig. 1.1.** Oscillatory behavior of the function (1.3) caused by cancellation errors

(the obvious result being 1 for any $x \neq 0$):

```
>> x =  1.e-15; ((1+x)-1)/x
ans =
    1.1102
```

This result is rather imprecise, the relative error being larger than 11%!

Another case of numerical cancellation is encountered while evaluating the function

$$f(x) = x^7 - 7x^6 + 21x^5 - 35x^4 + 35x^3 - 21x^2 + 7x - 1 \qquad (1.3)$$

at 401 equispaced points with abscissa in $[1 - 2 \cdot 10^{-8}, 1 + 2 \cdot 10^{-8}]$. We obtain the chaotic graph reported in Figure 1.1 (the real behavior is that of $(x-1)^7$, which is substantially constant and equal to the null function in such a tiny neighborhood of $x = 1$). The MATLAB commands that have generated this graph will be illustrated in Section 1.5.

Finally, it is interesting to notice that in $\mathbb{F}$ there is no place for indeterminate forms such as $0/0$ or $\infty/\infty$. Their presence produces what is called *not a number* (`NaN` in MATLAB or in Octave), for which the normal rules of calculus do not apply.

`NaN`

**Remark 1.1** Whereas it is true that roundoff errors are usually small, when repeated within long and complex algorithms, they may give rise to catastrophic effects. Two outstanding cases concern the explosion of the Ariane missile on June 4, 1996, engendered by an overflow in the computer on board, and the failure of the mission of an American Patriot missile, during the Gulf War in 1991, because of a roundoff error in the computation of its trajectory.

An example with less catastrophic (but still troublesome) consequences is provided by the sequence

$$z_2 = 2,\ z_{n+1} = 2^{n-1/2}\sqrt{1 - \sqrt{1 - 4^{1-n} z_n^2}}, \quad n = 2, 3, \ldots \qquad (1.4)$$

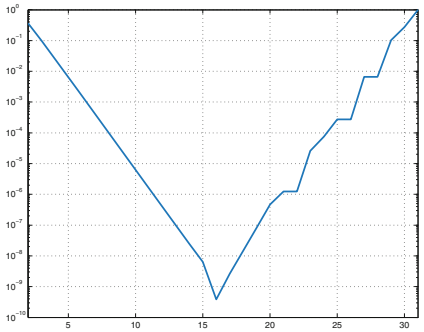

**Fig. 1.2.** Relative error $|\pi - z_n|/\pi$ versus $n$

which converges to $\pi$ when $n$ tends to infinity. When MATLAB is used to compute $z_n$, the relative error found between $\pi$ and $z_n$ decreases for the 16 first iterations, then grows because of roundoff errors (as shown in Figure 1.2). ■

See the Exercises 1.1-1.2.

## 1.3 Complex numbers

Complex numbers, whose set is denoted by $\mathbb{C}$, have the form $z = x + iy$, where $i = \sqrt{-1}$ is the imaginary unit (that is $i^2 = -1$), while $x = \mathrm{Re}(z)$ and $y = \mathrm{Im}(z)$ are the real and imaginary part of $z$, respectively. They are generally represented on the computer as pairs of real numbers.

Unless redefined otherwise, MATLAB variables `i` as well as `j` denote the imaginary unit. To introduce a complex number with real part `x` and imaginary part `y`, one can just write `x+i*y`; as an alternative, one can
complex use the command `complex(x,y)`. Let us also mention the exponential and the trigonometric representations of a complex number $z$, that are equivalent thanks to the *Euler formula*

$$z = \rho e^{i\theta} = \rho(\cos\theta + i\sin\theta); \tag{1.5}$$

$\rho = \sqrt{x^2 + y^2}$ is the modulus of the complex number (it can be obtained
abs by setting `abs(z)`) while $\theta$ is its argument, that is the angle between the $x$ axis and the straight line issuing from the origin and passing from the point of coordinate $x$, $y$ in the complex plane. $\theta$ can be found by typing
angle `angle(z)`. The representation (1.5) is therefore:

```
abs(z)*(cos(angle(z))+i*sin(angle(z))).
```

The graphical polar representation of one or more complex numbers
compass can be obtained through the command `compass(z)`, where `z` is either a single complex number or a vector whose components are complex numbers. For instance, by typing

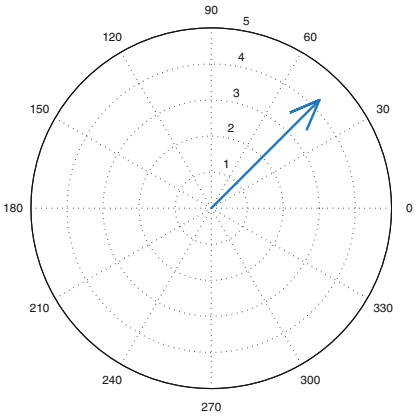


**Fig. 1.3.** Output of the MATLAB command `compass`

```
>> z = 3+i*3; compass(z);
```

one obtains the graph reported in Figure 1.3.

For any given complex number `z`, one can extract its real part with the command `real(z)` and its imaginary part with `imag(z)`. Finally, the complex conjugate $\bar{z} = x - iy$ of $z$, can be obtained by simply writing `conj(z)`.

`real` `imag`

`conj`

In MATLAB all operations are carried out by implicitly assuming that the operands as well as the result are complex. We may therefore find some apparently surprising results. For instance, if we compute the cube root of $-5$ with the MATLAB command `(-5)^(1/3)`, instead of $-1.7100\ldots$ we obtain the complex number $0.8550 + 1.4809i$. (We anticipate the use of the symbol `^` for the power exponent.) As a matter of fact, all numbers of the form $\rho e^{i(\theta+2k\pi)}$, with $k$ an integer, are indistinguishable from $z = \rho e^{i\theta}$. By computing the complex roots of $z$ of order three, we find $\sqrt[3]{\rho} e^{i(\theta/3+2k\pi/3)}$, that is, the three distinct roots

`^`

$$z_1 = \sqrt[3]{\rho} e^{i\theta/3}, \quad z_2 = \sqrt[3]{\rho} e^{i(\theta/3+2\pi/3)}, \quad z_3 = \sqrt[3]{\rho} e^{i(\theta/3+4\pi/3)}.$$

MATLAB will select the one that is encountered by spanning the complex plane counterclockwise beginning from the real axis. Since the polar representation of $z = -5$ is $\rho e^{i\theta}$ with $\rho = 5$ and $\theta = \pi$, the three roots are (see Figure 1.4 for their representation in the Gauss plane)

$$z_1 = \sqrt[3]{5}(\cos(\pi/3) + i\sin(\pi/3)) \simeq 0.8550 + 1.4809i,$$

$$z_2 = \sqrt[3]{5}(\cos(\pi) + i\sin(\pi)) \simeq -1.7100,$$

$$z_3 = \sqrt[3]{5}(\cos(-\pi/3) + i\sin(-\pi/3)) \simeq 0.8550 - 1.4809i.$$

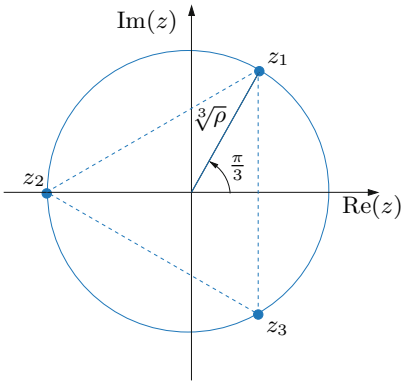


**Fig. 1.4.** Representation in the complex plane of the three complex cube roots of the real number $-5$

The first root is the one which is selected.

Finally, by (1.5) we obtain

$$\cos(\theta) = \frac{1}{2}\left(e^{i\theta} + e^{-i\theta}\right), \quad \sin(\theta) = \frac{1}{2i}\left(e^{i\theta} - e^{-i\theta}\right). \tag{1.6}$$

## 1.4 Matrices

Let $n$ and $m$ be positive integers. A matrix with $m$ rows and $n$ columns is a set of $m \times n$ elements $a_{ij}$, with $i = 1, \ldots, m$, $j = 1, \ldots, n$, represented by the following table:

$$\mathrm{A} = \begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{bmatrix}. \tag{1.7}$$

In compact form we write $\mathrm{A} = (a_{ij})$. Should the elements of A be real numbers, we write $\mathrm{A} \in \mathbb{R}^{m\times n}$, and $\mathrm{A} \in \mathbb{C}^{m\times n}$ if they are complex.

Square matrices of dimension $n$ are those with $m = n$. A matrix featuring a single column is a *column vector*, whereas a matrix featuring a single row is a *row vector*.

In order to introduce a matrix in MATLAB one has to write the elements from the first to the last row, introducing the character ; to separate the different rows. For instance, the command

```
>> A = [ 1 2 3; 4 5 6]
```

produces

```
A =
     1     2     3
     4     5     6
```

that is, a $2 \times 3$ matrix whose elements are indicated above. The $m \times n$ matrix `zeros(m,n)` has all null entries, `eye(m,n)` has all null entries unless $a_{ii}$, $i = 1, \ldots, \min(m, n)$, on the diagonal that are all equal to 1. The $n \times n$ identity matrix is obtained with the command `eye(n)` (which is an abridged version of `eye(n,n)`): its elements are $\delta_{ij} = 1$ if $i = j$, 0 otherwise, for $i, j = 1, \ldots, n$. Finally, by the command `A=[ ]` we can initialize an empty matrix.

zeros
eye
[ ]

We recall the following matrix operations:

1. if $\mathrm{A} = (a_{ij})$ and $\mathrm{B} = (b_{ij})$ are $m \times n$ matrices, the *sum* of A and B is the matrix $\mathrm{A} + \mathrm{B} = (a_{ij} + b_{ij})$;
2. the *product* of a matrix A by a real or complex number $\lambda$ is the matrix $\lambda\mathrm{A} = (\lambda a_{ij})$;
3. the *product* of two matrices is possible only for compatible sizes, precisely if A is $m \times p$ and B is $p \times n$, for some positive integer $p$. In that case $\mathrm{C} = \mathrm{AB}$ is an $m \times n$ matrix whose elements are

$$c_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj}, \quad \text{for } i = 1, \ldots, m, \ j = 1, \ldots, n.$$

Here is an example of the sum and product of two matrices.

```
>> A=[1 2 3; 4 5 6];
>> B=[7 8 9; 10 11 12];
>> C=[13 14; 15 16; 17 18];
>> A+B
 ans =
     8    10    12
    14    16    18
>> A*C
 ans =
    94   100
   229   244
```

Note that MATLAB returns a diagnostic message when one tries to carry out operations on matrices with incompatible dimensions. For instance:

```
>> A=[1 2 3; 4 5 6];
>> B=[7 8 9; 10 11 12];
>> C=[13 14; 15 16; 17 18];

>> A+C
 ??? Error using ==> +
 Matrix dimensions must agree.
>> A*B
 ??? Error using ==> *
 Inner matrix dimensions must agree.
```

If A is a square matrix of dimension $n$, its *inverse* (provided it exists) is a square matrix of dimension $n$, denoted by $\mathrm{A}^{-1}$, which satisfies the matrix relation $\mathrm{A}\mathrm{A}^{-1} = \mathrm{A}^{-1}\mathrm{A} = \mathrm{I}$. We can obtain $\mathrm{A}^{-1}$ through the command `inv(A)`. The inverse of A exists iff the *determinant* of A, a number denoted by det(A) and computed by the command `det(A)`, is non-zero. The latter condition is satisfied iff the column vectors of A are linearly independent (see Section 1.4.1). The determinant of a square matrix is defined by the following recursive formula (*Laplace rule*):

$$\det(\mathrm{A}) = \begin{cases} a_{11} & \text{if } n = 1, \\ \displaystyle\sum_{j=1}^{n} \Delta_{ij} a_{ij}, \text{ for } n > 1, \ \forall i = 1, \ldots, n, \end{cases} \tag{1.8}$$

where $\Delta_{ij} = (-1)^{i+j} \det(\mathrm{A}_{ij})$ and $\mathrm{A}_{ij}$ is the matrix obtained by eliminating the $i$-th row and $j$-th column from matrix A. (The result is independent of the row index $i$.) In particular, if $\mathrm{A} \in \mathbb{R}^{2\times 2}$ one has

$$\det(\mathrm{A}) = a_{11}a_{22} - a_{12}a_{21},$$

while if $\mathrm{A} \in \mathbb{R}^{3\times 3}$ we obtain

$$\det(\mathrm{A}) = a_{11}a_{22}a_{33} + a_{31}a_{12}a_{23} + a_{21}a_{13}a_{32}$$
$$-a_{11}a_{23}a_{32} - a_{21}a_{12}a_{33} - a_{31}a_{13}a_{22}.$$

We recall that if $\mathrm{A} = \mathrm{BC}$, then $\det(\mathrm{A}) = \det(\mathrm{B})\det(\mathrm{C})$.

To invert a $2 \times 2$ matrix and compute its determinant we can proceed as follows:

```
>> A=[1 2; 3 4];
>> inv(A)
 ans =
    -2.0000    1.0000
     1.5000   -0.5000
>> det(A)
 ans =
     -2
```

Should a matrix be singular, MATLAB returns a diagnostic message, followed by a matrix whose elements are all equal to `Inf`, as illustrated by the following example:

```
>> A=[1 2; 0 0];
>> inv(A)
 Warning: Matrix is singular to working precision.
 ans =
    Inf    Inf
    Inf    Inf
```

For special classes of square matrices, the computation of inverses and determinants is rather simple. In particular, if A is a *diagonal matrix*, i.e.

one for which only the diagonal elements $a_{kk}$, $k = 1, \ldots, n$, are non-zero, its determinant is given by $\det(\mathrm{A}) = a_{11}a_{22}\cdots a_{nn}$. In particular, A is non-singular iff $a_{kk} \neq 0$ for all $k$. In such a case the inverse of A is still a diagonal matrix with elements $a_{kk}^{-1}$.

Let `v` be a vector of dimension `n`. The command `diag(v)` produces a diagonal matrix whose elements are the components of vector `v`. The more general command `diag(v,m)` yields a square matrix of dimension `n+abs(m)` whose $m$-th upper diagonal (i.e. the diagonal made of elements with indices $i, i+m$) has elements equal to the components of `v`, while the remaining elements are null. Note that this extension is valid also when `m` is negative, in which case the only affected elements are those of lower diagonals. diag

For instance if `v = [1 2 3]` then:

```
>> A=diag(v,-1)
  A =
       0       0       0       0
       1       0       0       0
       0       2       0       0
       0       0       3       0
```

Other special cases are the *upper triangular* and *lower triangular* matrices. A square matrix of dimension $n$ is *lower* (respectively, *upper*) *triangular* if all elements above (respectively, below) the main diagonal are zero. Its determinant is simply the product of the diagonal elements.

Through the commands `tril(A)` and `triu(A)`, one can extract from the matrix `A` of dimension `n` its lower and upper triangular part. Their extensions `tril(A,m)` or `triu(A,m)`, with `m` ranging from `-n` and `n`, allow the extraction of the triangular part augmented by, or deprived of, extradiagonals. tril triu

For instance, given the matrix `A =[3 1 2; -1 3 4; -2 -1 3]`, by the command `L1=tril(A)` we obtain

```
L1 =
      3       0       0
     -1       3       0
     -2      -1       3
```

while, by `L2=tril(A,1)`, we obtain

```
L2 =
      3       1       0
     -1       3       4
     -2      -1       3
```

We recall that if $\mathrm{A} \in \mathbb{R}^{m\times n}$ its transpose $\mathrm{A}^T \in \mathbb{R}^{n\times m}$ is the matrix obtained by interchanging rows and columns of A. When $n = m$ and $\mathrm{A} = \mathrm{A}^T$ the matrix A is called *symmetric*. Finally, `A'` denotes the transpose of `A` if `A` is real, or its conjugate transpose (that is, $\mathrm{A}^H$) if `A` is complex. A square complex matrix that coincides with its conjugate transpose $\mathrm{A}^H$ is called *hermitian*. A'

**Octave 1.1** Also Octave returns a diagnostic message when one tries to carry out operations on matrices having non-compatible dimensions. If we repeat the previous MATLAB examples we obtain:

```
octave:1> A=[1 2 3; 4 5 6];
octave:2> B=[7 8 9; 10 11 12];
octave:3> C=[13 14; 15 16; 17 18];
octave:4> A+C
   error: operator +: nonconformant arguments (op1 is
   2x3, op2 is 3x2)
   error: evaluating binary operator ‘+’ near line 2,
   column 2
octave:5> A*B
   error: operator *: nonconformant arguments (op1 is
   x3, op2 is 2x3)
   error: evaluating binary operator ‘*’ near line 2,
   column 2
```

If A is singular, Octave returns a diagnostic message followed by the matrix whose elements are all equal to `Inf`, as illustrated by the following example:

```
octave:1> A=[1 2; 0 0];
octave:2> inv(A)
   warning: inverse: matrix singular to machine
   precision, rcond = 0
   ans =
      Inf   Inf
      Inf   Inf
```

■

### 1.4.1 Vectors

Vectors will be indicated in boldface; precisely, $\mathbf{v}$ will denote a column vector whose $i$-th component is denoted by $v_i$. When all components are real numbers we can write $\mathbf{v} \in \mathbb{R}^n$.

In MATLAB, vectors are regarded as particular cases of matrices. To introduce a column vector one has to insert between square brackets the values of its components separated by semi-colons, whereas for a row vector it suffices to write the component values separated by blanks or commas. For instance, through the instructions `v = [1;2;3]` and `w = [1 2 3]` we initialize the column vector $\mathbf{v}$ and the row vector $\mathbf{w}$, both of dimension 3. The command `zeros(n,1)`(respectively, `zeros(1,n)`) produces a column (respectively, row) vector of dimension `n` with null elements, which we will denote by $\mathbf{0}$. Similarly, the command `ones(n,1)` generates the column vector, denoted with $\mathbf{1}$, whose components are all equal to 1.

ones

A system of vectors $\{\mathbf{y}_1, \ldots, \mathbf{y}_m\}$ is *linearly independent* if the relation

$$\alpha_1 \mathbf{y}_1 + \ldots + \alpha_m \mathbf{y}_m = \mathbf{0}$$

implies that all coefficients $\alpha_1, \ldots, \alpha_m$ are null. A system $\mathcal{B} = \{\mathbf{y}_1, \ldots, \mathbf{y}_n\}$ of $n$ linearly independent vectors in $\mathbb{R}^n$ (or $\mathbb{C}^n$) is a *basis* for $\mathbb{R}^n$ (or $\mathbb{C}^n$), that is, any vector $\mathbf{w}$ in $\mathbb{R}^n$ can be written as a linear combination of the elements of $\mathcal{B}$,

$$\mathbf{w} = \sum_{k=1}^{n} w_k \mathbf{y}_k,$$

for a unique possible choice of the coefficients $\{w_k\}$. The latter are called the *components* of $\mathbf{w}$ with respect to the basis $\mathcal{B}$. For instance, the canonical basis of $\mathbb{R}^n$ is the set of vectors $\{\mathbf{e}_1, \ldots, \mathbf{e}_n\}$, where $\mathbf{e}_i$ has its $i$-th component equal to 1, and all other components equal to 0 and is the one which is normally used.

The *scalar product* of two vectors $\mathbf{v}, \mathbf{w} \in \mathbb{R}^n$ is defined as

$$(\mathbf{v}, \mathbf{w}) = \mathbf{w}^T \mathbf{v} = \sum_{k=1}^{n} v_k w_k,$$

$\{v_k\}$ and $\{w_k\}$ being the components of $\mathbf{v}$ and $\mathbf{w}$, respectively. The corresponding command is `w'*v` or else `dot(v,w)`, where now the apex denotes transposition of the vector. For a vector $\mathbf{v}$ with complex components, `v'` denotes its conjugate transpose $\mathbf{v}^H$, that is a row-vector whose components are the complex conjugate $\bar{v}_k$ of $v_k$. The length (or modulus) of a vector $\mathbf{v}$ is given by

dot

v'

$$\|\mathbf{v}\| = \sqrt{(\mathbf{v}, \mathbf{v})} = \sqrt{\sum_{k=1}^{n} v_k^2}$$

and can be computed through the command `norm(v)`; $\|\mathbf{v}\|$ is also said *euclidean norm* of the vector $\mathbf{v}$.

norm

The *vector product* between two vectors $\mathbf{v}, \mathbf{w} \in \mathbb{R}^3$, $\mathbf{v} \times \mathbf{w}$ or $\mathbf{v} \wedge \mathbf{w}$, is the vector $\mathbf{u} \in \mathbb{R}^3$ orthogonal to both $\mathbf{v}$ and $\mathbf{w}$ whose modulus is $|\mathbf{u}| = |\mathbf{v}|\ |\mathbf{w}| \sin(\alpha)$, where $\alpha$ is the smaller angle formed by $\mathbf{v}$ and $\mathbf{w}$. It can be obtained by the command `cross(v,w)`.

cross

The visualization of a vector can be obtained by the MATLAB command `quiver` in $\mathbb{R}^2$ and `quiver3` in $\mathbb{R}^3$.

quiver

quiver3

The MATLAB command `x.*y`, `x./y` or `x.^2` indicates that these operations should be carried out component by component. For instance if we define the vectors

.*  ./  .^

```
>> x = [1; 2; 3]; y = [4; 5; 6];
```

the instruction

```
>> y'*x
ans =
     32
```

provides their scalar product, while

```
>> x.*y
 ans =
       4
      10
      18
```

returns a vector whose $i$-th component is equal to $x_i y_i$.

Finally, we recall that a vector $\mathbf{v} \in \mathbb{C}^n$, with $\mathbf{v} \neq \mathbf{0}$, is an *eigenvector* of a matrix $\mathrm{A} \in \mathbb{C}^{n \times n}$ associated with the complex number $\lambda$ if

$$\mathrm{A}\mathbf{v} = \lambda \mathbf{v}.$$

The complex number $\lambda$ is called *eigenvalue* of A. In general, the computation of eigenvalues is quite difficult. Exceptions are represented by diagonal and triangular matrices, whose eigenvalues are their diagonal elements.

See the Exercises 1.3-1.6.

## 1.5 Real functions

This section deals with manipulation of real functions. More particularly, for a given function $f$ defined on an interval $(a, b)$, we aim at computing its zeros, its integral and its derivative, as well as drawing its graph. The

fplot

command `fplot(fun,lims)` plots the graph of the function `fun` (which is stored as a string of characters) on the interval (`lims(1)`, `lims(2)`). For instance, to represent $f(x) = 1/(1 + x^2)$ on the interval $(-5, 5)$, we can write

```
>> fun ='1/(1+x^2)'; lims=[-5,5]; fplot(fun,lims);
```

or, more directly,

```
>> fplot('1/(1+x^2)',[-5 5]);
```

In MATLAB the graph is obtained by sampling the function on a set of non-equispaced abscissae and reproduces the true graph of $f$ with a tolerance of 0.2%. To improve the accuracy we could use the command

```
>> fplot(fun,lims,tol,n,LineSpec)
```

where `tol` indicates the desired tolerance and the parameter `n`($\geq 1$) ensures that the function will be plotted with a minimum of `n`$+1$ points.

`LineSpec` is a string specifying the style or the color of the line used for plotting the graph. For example, `LineSpec='--'` is used for a dashed line, `LineSpec='r-.'` for a red dashed-dotted line, etc. To use default values for `tol`, `n` or `LineSpec` one can pass empty matrices (`[ ]`).
By writing `grid on` after the command `fplot`, we can obtain the background-grid as that in Figure 1.1. `grid`
The function $f(x) = 1/(1+x^2)$ can be defined in several different ways:
by the instruction `fun='1/(1+x^2)'` seen before;
by the command `inline` with the instruction `inline`

```
>> fun=inline('1/(1+x^2)','x');
```

by *anonymous function* and the use of a *function handle* `@` as follows `@`

```
>> fun=@(x)[1/(1+x^2)];
```

finally, by writing a suitable MATLAB *function*:

```
function y=fun(x)
y=1/(1+x^2);
end
```

The `inline` command, whose common syntax is
`fun=inline(expr, arg1, arg2, ..., argn)`,
defines a *function* `fun` depending on the ordered set of variables `arg1, arg2, ..., argn`. The string `expr` contains the expression of `fun`. For example, `fun=inline('sin(x)*(1+cos(t))','x','t')` defines the function $fun(x,t) = \sin(x)(1+\cos(t))$. The brief form `fun=inline(expr)` implicitely supposes that `expr` depends on all the variables which appear in the definition of the function itself, by following alphabetical order. For example, by the command `fun=inline('sin(x)*(1+cos(t))')` we define the function $fun(t,x) = \sin(x)(1+\cos(t))$, whose first variable is $t$, while the second one is $x$ (by following lexicographical order).
The common syntax of an *anonymous function* reads
`fun=@(arg1, arg2,...,argn)[expr]`.

In order to evaluate the function `fun` at a point `x` (or at a set of points, stored in the vector `x`) we can make use of the commands `eval`, or `feval`, otherwise we can simply evaluate the function consistently with the command used to define the function itself. Even if they produce the same result, the commands `eval` and `feval` have a different syntax. `eval` has only one input parameter (the name of the mathematical function to be evaluated) and evaluates the function `fun` at the point stored in the variable which appears inside the definition of `fun` (i.e., `x` in the above definitions). On the contrary, the function `feval` has at least two parameters; the former is the name `fun` of the mathematical function to be evaluated, the latter contains the inputs to the function `fun`. `eval` `feval`

We report in Table 1.1 the various ways for defining, evaluating and plotting a mathematical function. In the following, we will use one of

| Definition | Evaluation | Plotting |
|---|---|---|
| `fun='1/(1+x^2)'` | `y=eval(fun)` | `fplot(fun,[-2,2])`<br>`fplot('fun',[-2,2])` |
| `fun=inline('1/(1+x^2)')` | `y=fun(x)`<br>`y=feval(fun,x)`<br>`y=feval('fun',x)` | `fplot(fun,[-2,2])`<br>`fplot('fun',[-2,2])` |
| `fun=@(x)[1/(1+x^2)]` | `y=fun(x)`<br>`y=feval(fun,x)`<br>`y=feval('fun',x)` | `fplot(fun,[-2,2])`<br>`fplot('fun',[-2,2])` |
| `function y=fun(x)`<br>`y=1/(1+x^2);`<br>`end` | `y=fun(x)`<br>`y=feval(@fun,x)`<br>`y=feval('fun',x)` | `fplot('fun',[-2,2])`<br>`fplot(@fun,[-2,2])` |

**Table 1.1.** How to define, evaluate and plot a mathematical function

the definitions of Table 1.1 and proceed coherently. However, the reader could make different choices.

If the variable `x` is an array, the operations `/, * and ^` acting on arrays have to be replaced by the corresponding *dot operations* `./`, `.*` and `.^` which operate component-wise. For instance, the instruction `fun=@(x)[1/(1+x^2)]` is replaced by `fun=@(x)[1./(1+x.^2)]`.

plot The command `plot` can be used as alternative to `fplot`, provided that the mathematical function has been evaluated on a set of abscissa. The following instructions

```
>> x=linspace(-2,3,100);
>> y=exp(x).*(sin(x).^2)-0.4;
>> plot(x,y,'c','Linewidth',2); grid on
```

linspace produce a graph in linear scale, precisely the command `linspace(a,b,n)` generates a row array of $n$ equispaced points from $a$ to $b$, while the command `plot(x,y,'c','Linewidth',2)` creates a linear piecewise curve connecting the points $(x_i, y_i)$ (for $i = 1, \ldots, n$) with a cyan line width of 2 points.

### 1.5.1 The zeros

We recall that if $f(\alpha) = 0$, $\alpha$ is called *zero* of $f$ or *root* of the equation $f(x) = 0$. A zero is *simple* if $f'(\alpha) \neq 0$, *multiple* otherwise.

From the graph of a function one can infer (within a certain tolerance) which are its real zeros. The direct computation of all zeros of a given function is not always possible. For functions which are polynomials with real coefficients of degree $n$, that is, of the form

$$p_n(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n = \sum_{k=0}^{n} a_k x^k, \quad a_k \in \mathbb{R},\ a_n \neq 0,$$

we can obtain the only zero $\alpha = -a_0/a_1$, when $n = 1$ (i.e. $p_1$ represents a straight line), or the two zeros, $\alpha_+$ and $\alpha_-$, when $n = 2$ (this time $p_2$ represents a parabola) $\alpha_\pm = (-a_1 \pm \sqrt{a_1^2 - 4a_0 a_2})/(2a_2)$.

However, there are no explicit formulae for the zeros of an arbitrary polynomial $p_n$ when $n \geq 5$.

In what follows we will denote with $\mathbb{P}_n$ the space of polynomials of degree less than or equal to $n$,

$$p_n(x) = \sum_{k=0}^{n} a_k x^k \tag{1.9}$$

where the $a_k$ are given coefficients, real or complex.

Also the number of zeros of a function cannot in general be determined *a priori*. An exception is provided by polynomials, for which the number of zeros (real or complex) coincides with the polynomial degree. Moreover, should $\alpha = x + iy$ with $y \neq 0$ be a zero of a polynomial with degree $n \geq 2$, if $a_k$ are real coefficients, then its complex conjugate $\bar{\alpha} = x - iy$ is also a zero.

To compute in MATLAB one zero of a function `fun`, near a given value `x0`, either real or complex, the command `fzero(fun,x0)` can be used. The result is an approximate value of the desired zero, and also the interval in which the search was made. Alternatively, using the command `fzero(fun,[x0 x1])`, a zero of `fun` is searched for in the interval whose endpoints are `x0,x1`, provided $f$ changes sign between `x0` and `x1`. fzero

Let us consider, for instance, the function $f(x) = x^2 - 1 + e^x$. Looking at its graph we see that there are two zeros in $(-1, 1)$. To compute them we need to execute the following commands:

```
>> fun=@(x)[x^2 - 1 + exp(x)];
>> fzero(fun,-1)
 ans =
    -0.7146

>> fzero(fun,1)
 ans =
    5.4422e-18
```

Alternatively, after noticing from the function plot that one zero is in the interval $[-1, -0.2]$ and another in $[-0.2, 1]$, we could have written

```
>> fzero(fun,[-1  -0.2])
ans =
   -0.7146
```

```
>> fzero(fun,[-0.2 1])
ans =
  -5.2609e-17
```

The result obtained for the second zero is slightly different than the one obtained previously, due to a different initialization of the algorithm implemented in `fzero`. In Chapter 2 we will introduce and investigate several methods for the approximate computation of the zeros of an arbitrary function.

The `fzero` syntax is the same if the function `fun` is defined either by the command `inline` or by a string.

Otherwise, if `fun` is defined by an M-file, we can choose one between these two calls:

```
>> fzero('fun', 1)
```

or

```
>> fzero(@fun,1)
```

**Octave 1.2** In Octave the function `fzero` accepts as input mathematical functions defined with either `inline`, anonymous function or M-file functions. ■

### 1.5.2 Polynomials

polyval

Polynomials are very special functions and there is a special MATLAB toolbox `polyfun` for their treatment. The command `polyval` is apt to evaluate a polynomial at one or several points. Its input arguments are a vector `p` and a vector `x`, where the components of `p` are the polynomial coefficients stored in decreasing order, from $a_n$ down to $a_0$, and the components of `x` are the abscissae where the polynomial needs to be evaluated. The result can be stored in a vector `y` by writing

```
>> y = polyval(p,x)
```

For instance, the values of $p(x) = x^7+3x^2-1$, at the equispaced abscissae $x_k = -1+k/4$ for $k = 0,\dots,8$, can be obtained by proceeding as follows:

```
>> p = [1 0 0 0 0 3 0 -1]; x = [-1:0.25:1];
>> y = polyval(p,x)
y =
  Columns 1 through 5:
    1.00000   0.55402  -0.25781  -0.81256  -1.00000
  Columns 6 through 9:
    -0.81244  -0.24219   0.82098   3.00000
```

Alternatively, one could use the command `feval`. However, in such case one should provide the entire analytic expression of the polynomial in the input string, and not simply its coefficients.

The program `roots` provides an approximation of the zeros of a polynomial and requires only the input of the vector `p`. For instance, we can compute the zeros of $p(x) = x^3 - 6x^2 + 11x - 6$ by writing

roots

```
>> p = [1 -6 11 -6]; format long;
>> roots(p)

ans =
   3.00000000000000
   2.00000000000000
   1.00000000000000
```

Unfortunately, the result is not always that accurate. For instance, for the polynomial $p(x) = (x+1)^7$, whose unique zero is $\alpha = -1$ with multiplicity 7, we find (quite surprisingly)

```
>> p = [1 7  21 35  35  21  7  1];
>> roots(p)

ans =
  -1.0101
  -1.0063 + 0.0079i
  -1.0063 - 0.0079i
  -0.9977 + 0.0099i
  -0.9977 - 0.0099i
  -0.9909 + 0.0044i
  -0.9909 - 0.0044i
```

In fact, numerical methods for the computation of the polynomial roots with multiplicity larger than one are particularly subject to round-off errors (see Section 2.6.2).

The command `p=conv(p1,p2)` returns the coefficients of the polynomial given by the product of two polynomials whose coefficients are contained in the vectors `p1` and `p2`.

conv

Similarly, the command `[q,r]=deconv(p1,p2)` provides the coefficients of the polynomials obtained on dividing `p1` by `p2`, i.e. `p1 = conv(p2,q) + r`. In other words, `q` and `r` are the quotient and the remainder of the division.

deconv

Let us consider for instance the product and the ratio between the two polynomials $p_1(x) = x^4 - 1$ and $p_2(x) = x^3 - 1$ :

```
>> p1 = [1 0 0 0 -1];
>> p2 = [1 0 0 -1];
>> p=conv(p1,p2)

p =
     1     0     0    -1    -1     0     0     1
```

| command | yields |
|---|---|
| `y=polyval(p,x)` | `y` = values of $p(x)$ |
| `z=roots(p)` | `z` = roots of $p$ such that $p(z)=0$ |
| `p=conv(p1,p2)` | `p` = coefficients of the polynomial $p_1 p_2$ |
| `[q,r]=deconv(p1,p2)` | `q` = coefficients of $q$, `r` = coefficients of $r$ such that $p_1 = qp_2 + r$ |
| `y=polyder(p)` | `y` = coefficients of $p'(x)$ |
| `y=polyint(p)` | `y` = coefficients of $\int_0^x p(t)\,dt$ |

**Table 1.2.** MATLAB commands for polynomial operations

```
>> [q,r]=deconv(p1,p2)

q =
     1     0
r =
     0     0     0     1    -1
```

We therefore find the polynomials $p(x) = p_1(x)p_2(x) = x^7 - x^4 - x^3 + 1$, $q(x) = x$ and $r(x) = x - 1$ such that $p_1(x) = q(x)p_2(x) + r(x)$.

polyint polyder

The commands `polyint(p)` and `polyder(p)` provide respectively the coefficients of the primitive (vanishing at $x = 0$) and those of the derivative of the polynomial whose coefficients are given by the components of the vector `p`.

If `x` is a vector of abscissae and `p` (respectively, $\mathtt{p}_1$ and $\mathtt{p}_2$) is a vector containing the coefficients of a polynomial $p$ (respectively, $p_1$ and $p_2$), the previous commands are summarized in Table 1.2.

polyfit

A further command, `polyfit`, allows the computation of the $n+1$ polynomial coefficients of a polynomial $p$ of degree $n$ once the values attained by $p$ at $n+1$ distinct nodes are available (see Section 3.3.1).

### 1.5.3 Integration and differentiation

The following two results will often be invoked throughout this book:

1. the *fundamental theorem of integration*: if $f$ is a continuous function in $[a, b)$, then

$$F(x) = \int_a^x f(t)\,dt \qquad \forall x \in [a, b),$$

is a differentiable function, called a *primitive* of $f$, which satisfies,

$$F'(x) = f(x) \qquad \forall x \in [a, b);$$

2. the *first mean-value theorem for integrals*: if $f$ is a continuous function in $[a, b)$ and $x_1, x_2 \in [a, b)$ with $x_1 < x_2$, then $\exists \xi \in (x_1, x_2)$ such that

$$f(\xi) = \frac{1}{x_2 - x_1} \int_{x_1}^{x_2} f(t)\, dt.$$

Even when it does exist, a primitive might be either impossible to determine or difficult to compute. For instance, knowing that $\ln|x|$ is a primitive of $1/x$ is irrelevant if one doesn't know how to efficiently compute the logarithms. In Chapter 4 we will introduce several methods to compute the integral of an arbitrary continuous function with a desired accuracy, irrespectively of the knowledge of its primitive.

We recall that a function $f$ defined on an interval $[a, b]$ is differentiable in a point $\bar{x} \in (a, b)$ if the following limit exists and is finite

$$f'(\bar{x}) = \lim_{h \to 0} \frac{1}{h}(f(\bar{x} + h) - f(\bar{x})). \tag{1.10}$$

The value of $f'(\bar{x})$ provides the slope of the tangent line to the graph of $f$ at the point $\bar{x}$.

We say that a function which is continuous together with its derivative at any point of $[a, b]$ belongs to the space $C^1([a, b])$. More generally, a function with continuous derivatives up to the order $p$ (a positive integer) is said to belong to $C^p([a, b])$. In particular, $C^0([a, b])$ denotes the space of continuous functions in $[a, b]$.

A result that will be often used is the *mean-value theorem*, according to which, if $f \in C^1([a, b])$, there exists $\xi \in (a, b)$ such that

$$f'(\xi) = (f(b) - f(a))/(b - a).$$

Finally, it is worth recalling that a function that is continuous with all its derivatives up to the order $n$ in a neighborhood of $x_0$, can be approximated in such a neighborhood by the so-called *Taylor polynomial of degree n* at the point $x_0$:

$$\begin{aligned} T_n(x) &= f(x_0) + (x - x_0)f'(x_0) + \ldots + \frac{1}{n!}(x - x_0)^n f^{(n)}(x_0) \\ &= \sum_{k=0}^{n} \frac{(x - x_0)^k}{k!} f^{(k)}(x_0). \end{aligned}$$

The MATLAB toolbox `symbolic` provides the commands `diff`, `int` and `taylor` which allow us to obtain the analytical expression of the derivative, the indefinite integral (i.e. a primitive) and the Taylor polynomial, respectively, of a given function. In particular, having defined in the string `f` the function on which we intend to operate, `diff(f,n)`

diff int taylor

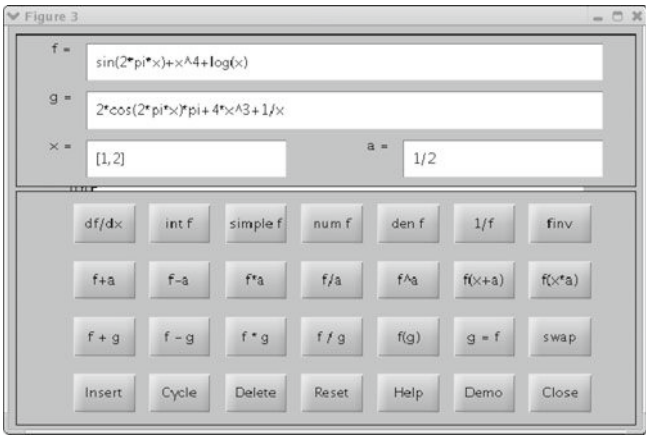


**Fig. 1.5.** Graphical interface of the command `funtool`

provides its derivative of order `n`, `int(f)` its indefinite integral, and `taylor(f,x,n+1)` the associated Taylor polynomial of degree `n` in a neighborhood of $x_0 = 0$. The variable `x` must be declared *symbolic* by using the command `syms x`. This will allow its algebraic manipulation without specifying its value.

syms

In order to do this for the function $f(x) = (x^2+2x+2)/(x^2-1)$, we proceed as follows:

```
>> f = '(x^2+2*x+2)/(x^2-1)';
>> syms x
>> diff(f)
 (2*x+2)/(x^2-1)-2*(x^2+2*x+2)/(x^2-1)^2*x
>> int(f)
 x+5/2*log(x-1)-1/2*log(1+x)
>> taylor(f,x,6)
 -2-2*x-3*x^2-2*x^3-3*x^4-2*x^5
```

simple

We observe that using the command `simple` it is possible to simplify the expressions generated by `diff`, `int` and `taylor` in order to make them as simple as possible. The command `funtool`, by the graphical interface illustrated in Fig. 1.5, allows a very easy symbolic manipulation of arbitrary functions.

funtool

**Octave 1.3** In Octave symbolic calculations can be performed by the Octave-Forge Symbolic package. Note, however, that the syntax of Octave-Forge is not in general compatible with that of the MATLAB `symbolic` toolbox. ■

See the Exercises 1.7-1.8.

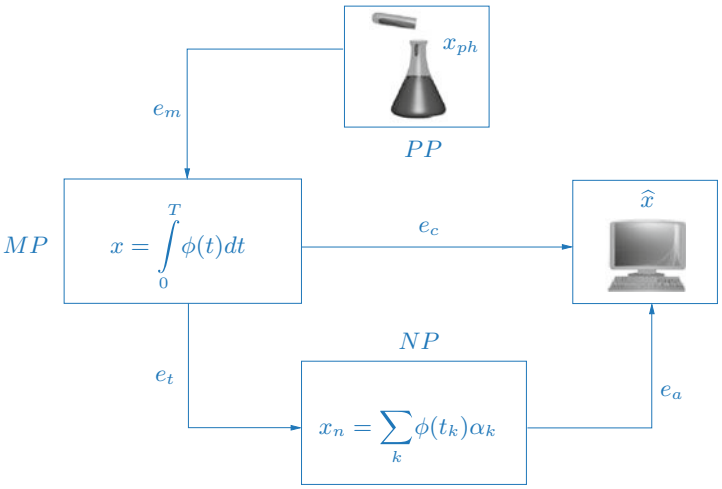


**Fig. 1.6.** Types of errors in a computational process

## 1.6 To err is not only human

As a matter of fact, by re-phrasing the Latin motto *errare humanum est*, we might say that in numerical computation to err is even inevitable.

As we have seen, the simple fact of using a computer to represent real numbers introduces errors. What is therefore important is not to strive to eliminate errors, but rather to be able to control their effect.

Generally speaking, we can identify several levels of errors that occur during the approximation and resolution of a physical problem (see Figure 1.6).

At the highest level stands the error $e_m$ which occurs when forcing the physical reality ($PP$ stands for physical problem and $x_{ph}$ denotes its solution) to obey some mathematical model ($MP$, whose solution is $x$). Such errors will limit the applicability of the mathematical model to certain situations and are beyond the control of Scientific Computing.

The mathematical model (whether expressed by an integral as in the example of Figure 1.6, an algebraic or differential equation, a linear or nonlinear system) is generally not solvable in explicit form. Its resolution by computer algorithms will surely involve the introduction and propagation of roundoff errors at least. Let's call these errors $e_a$.

On the other hand, it is often necessary to introduce further errors since any procedure of the mathematical model involving an infinite sequence of arithmetic operations cannot be performed by the computer unless approximately. For instance the computation of the sum of a series will necessarily be accomplished in an approximate way by considering a suitable truncation.

It will therefore be necessary to introduce a numerical problem, $NP$, whose solution $x_n$ differs from $x$ by an error $e_t$ which is called *truncation error*. Such errors do not only occur in mathematical models that are already set in finite dimension (for instance, when solving a linear system). The sum of the errors $e_a$ and $e_t$ constitutes the *computational error* $e_c$, the quantity we are interested in.

The *absolute* computational error is the difference between $x$, the exact solution of the mathematical model, and $\widehat{x}$, the solution obtained at the end of the numerical process,

$$e_c^{abs} = |x - \widehat{x}|,$$

while (if $x \neq 0$) the *relative* computational error is

$$e_c^{rel} = |x - \widehat{x}|/|x|,$$

where $|\cdot|$ denotes the modulus, or other measure of size, depending on the meaning of $x$.

The numerical process is generally an approximation of the mathematical model obtained as a function of a discretization parameter, which we will refer to as $h$ and suppose positive. If, as $h$ tends to 0, the numerical process returns the solution of the mathematical model, we will say that the numerical process is *convergent*. Moreover, if the (absolute or relative) error can be bounded as a function of $h$ as

$$\boxed{e_c \leq Ch^p} \tag{1.11}$$

where $C$ is independent of $h$ and $p$ is a positive number, we will say that the method is *convergent of order* $p$. It is sometimes even possible to replace the symbol $\leq$ with $\simeq$, in the case where, besides the upper bound (1.11), a lower bound $C'h^p \leq e_c$ is also available ($C'$ being another constant independent of $h$ and $p$).

**Example 1.1** Suppose we approximate the derivative of a function $f$ at a point $\bar{x}$ with the incremental ratio that appears in (1.10). Obviously, if $f$ is differentiable at $\bar{x}$, the error committed by replacing $f'$ by the incremental ratio tends to 0 as $h \to 0$. However, as we will see in Section 4.2, the error can be considered as $Ch$ only if $f \in C^2$ in a neighborhood of $\bar{x}$. ■

While studying the convergence properties of a numerical procedure we will often deal with graphs reporting the error as a function of $h$ in a logarithmic scale, which shows $\log(h)$ on the abscissae axis and $\log(e_c)$ on the ordinates axis. The purpose of this representation is easy to see: if $e_c = Ch^p$ then $\log e_c = \log C + p \log h$. In logarithmic scale therefore $p$ represents the slope of the straight line $\log e_c$, so if we must compare two methods, the one presenting the greater slope will be the one with a higher order. (The slope will be $p = 1$ for first-order methods, $p = 2$

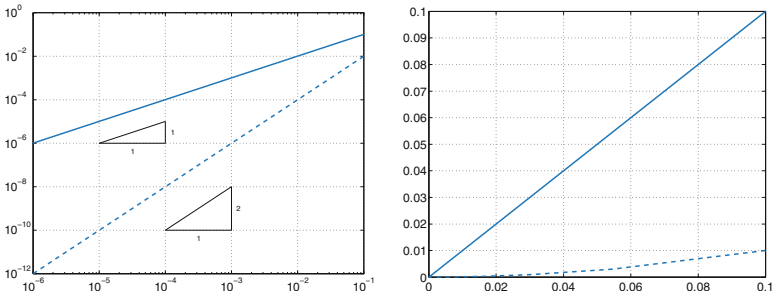


**Fig. 1.7.** Plot of the same data in log-log scale *(left)* and in linear-linear scale *(right)*

for second-order methods, and so on.) To obtain graphs in a logarithmic scale one just needs to type `loglog(x,y)`, `x` and `y` being the vectors containing the abscissae and the ordinates of the data to be represented. loglog

As an instance, in Figure 1.7, left, we report the straight lines relative to the behavior of the errors in two different methods. The continuous line represents a first-order approximation, while the dashed line represents a second-order one. In Figure 1.7, right, we show the same data plotted on the left, but now using the `plot` command, that is a linear scale for both $x-$ and $y-$ axis. It is evident that the linear representation of these data is not optimal, since the dashed curve appears thickened on the $x-$axis when $x \in [10^{-6}, 10^{-2}]$, even if the corresponding ordinates range from $10^{-12}$ to $10^{-4}$, spanning 8 orders of magnitude.

There is an alternative to the graphical way of establishing the order of a method when one knows the errors $e_i$ relative to some given values $h_i$ of the parameter of discretization, with $i = 1, \ldots, N$: it consists in conjecturing that $e_i$ is equal to $Ch_i^p$, where $C$ does not depend on $i$. One can then approach $p$ with the values:

$$p_i = \log(e_i/e_{i-1})/\log(h_i/h_{i-1}), \qquad i = 2, \ldots, N. \tag{1.12}$$

Actually the error is not a computable quantity since it depends on the unknown solution. Therefore it is necessary to introduce computable quantities that can be used to estimate the error itself, the so called *error estimator*. We will see some examples in Sections 2.3.1, 2.4 and 4.5.

Sometimes, instead of using the log-log scale, we will use the semi-logarithmic one, i.e. logarithmic scale on the $y$-axis and linear scale on the $x$-axis. This representation is preferable, for instance, in plotting the error of an iterative method versus the iterations, as done in Figure 1.2, or in general, when the ordinates span a wider interval than abscissae.

Let us consider the following 3 sequences, all converging to $\sqrt{2}$:

$$\begin{aligned} x_0 &= 1, \quad x_{n+1} = \frac{3}{4}x_n + \frac{1}{2x_n}, && n = 0, 1, \ldots, \\ y_0 &= 1, \quad y_{n+1} = \frac{1}{2}y_n + \frac{1}{y_n}, && n = 0, 1, \ldots, \\ z_0 &= 1, \quad z_{n+1} = \frac{3}{8}z_n + \frac{3}{2z_n} - \frac{1}{2z_n^3}, && n = 0, 1, \ldots. \end{aligned}$$

In Figure 1.8 we plot the errors $e_n^x = |x_n - \sqrt{2}|/\sqrt{2}$ (solid line), $e_n^y = |y_n - \sqrt{2}|/\sqrt{2}$ (dashed line) and $e_n^z = |z_n - \sqrt{2}|/\sqrt{2}$ (dashed-dotted line) versus iterations and in semi-logarithmic scale. It is possible to prove that

$$e_n^x \simeq \rho_x^n e_0^x, \quad e_n^y \simeq \rho_y^{n^2} e_0^y, \quad e_n^z \simeq \rho_z^{n^3} e_0^z,$$

where $\rho_x,\ \rho_y,\ \rho_z \in (0, 1)$, thus, by applying the logarithm only to the ordinates, we have

$$\begin{aligned} &\log(e_n^x) \simeq C_1 + \log(\rho_x)n, \qquad \log(e_n^y) \simeq C_2 + \log(\rho_y)n^2, \\ &\log(e_n^z) \simeq C_3 + \log(\rho_z)n^3, \end{aligned}$$

i.e., a straight line, a parabola and a cubic, respectively, exactly as we can see in Figure 1.8, left.

semilogy

The MATLAB command for semi-logharitmic scale is `semilogy(x,y)`, where `x` and `y` are arrays of the same size.

In Figure 1.8, right, we display the errors $e_n^x$, $e_n^y$ and $e_n^z$ versus iterations, in linear-linear scale and by using the command `plot`. It is evident that the use of semi-logarithmic instead of linear-linear scale is more appropriate.

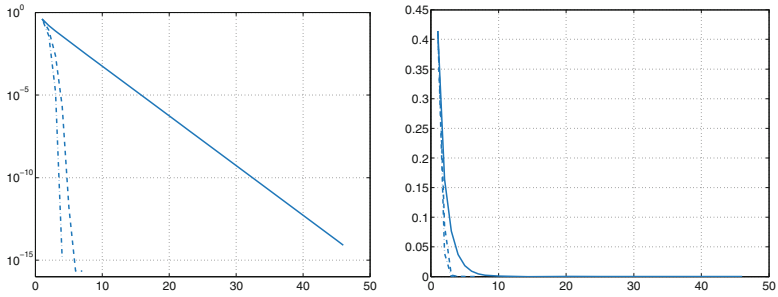


**Fig. 1.8.** Errors $e_n^x$ *(solid line)*, $e_n^y$ *(dashed line)* and $e_n^z$ *(dashed-dotted line)* in semi-logarithmic scale *(left)* and linear-linear scale *(right)*

### 1.6.1 Talking about costs

In general a problem is solved on the computer by an algorithm, which is a precise directive in the form of a finite text specifying the execution of a finite series of elementary operations. We are interested in those algorithms which involve only a finite number of steps.

The *computational cost* of an algorithm is the number of floating-point operations that are required for its execution. Often, the speed of a computer is measured by the maximum number of floating-point operations which the computer can execute in one second (*flops*). In particular, the following abridged notations are commonly used: Mega-flops, equal to $10^6$ *flops*, Giga-flops equal to $10^9$ *flops*, Tera-flops equal to $10^{12}$ *flops*, Peta-flops equal to $10^{15}$ *flops*. The fastest computers nowadays reach as many as 1.7 of Peta-flops.

In general, the exact knowledge of the number of operations required by a given algorithm is not essential. Rather, it is useful to determine its order of magnitude as a function of a parameter $d$ which is related to the problem dimension. We therefore say that an algorithm has *constant* complexity if it requires a number of operations independent of $d$, i.e. $\mathcal{O}(1)$ operations, *linear* complexity if it requires $\mathcal{O}(d)$ operations, or, more generally, *polynomial* complexity if it requires $\mathcal{O}(d^m)$ operations, for a positive integer $m$. Other algorithms may have *exponential* ($\mathcal{O}(c^d)$ operations) or even *factorial* ($\mathcal{O}(d!)$ operations) complexity. We recall that the symbol $\mathcal{O}(d^m)$ means "it behaves, for large $d$, like a constant times $d^m$".

**Example 1.2 (matrix-vector product)** Le A be a square matrix of order $n$ and let $\mathbf{v}$ be a vector of $\mathbb{R}^n$. The $j-th$ component of the product A$\mathbf{v}$ is given by

$$a_{j1}v_1 + a_{j2}v_2 + \ldots + a_{jn}v_n,$$

and requires $n$ products and $n-1$ additions. One needs therefore $n(2n-1)$ operations to compute all the components. Thus this algorithm requires $\mathcal{O}(n^2)$ operations, so it has a quadratic complexity with respect to the parameter $n$. The same algorithm would require $\mathcal{O}(n^3)$ operations to compute the product of two square matrices of order $n$. However, there is an algorithm, due to Strassen, which requires "only" $\mathcal{O}(n^{\log_2 7})$ operations and another, due to Winograd and Coppersmith, requiring $\mathcal{O}(n^{2.376})$ operations. ■

**Example 1.3 (computation of a matrix determinant)** As already mentioned, the determinant of a square matrix of order $n$ can be computed using the recursive formula (1.8). The corresponding algorithm has a factorial complexity with respect to $n$ and would be usable only for matrices of small dimension. For instance, if $n = 24$, a computer capable of performing as many as 1 Peta-flops (i.e. $10^{15}$ floating-point operations per second) would require 59 years to carry out this computation. One has therefore to resort to more efficient algorithms. Indeed, there exists an algorithm allowing the computation of

determinants through matrix-matrix products, with henceforth a complexity of $\mathcal{O}(n^{\log_2 7})$ operations by applying the Strassen algorithm previously mentioned (see [BB96]). ■

The number of operations is not the sole parameter which matters in the analysis of an algorithm. Another relevant factor is represented by the time that is needed to access the computer memory (which depends on the way the algorithm has been coded). An indicator of the performance of an algorithm is therefore the CPU time (CPU stands for *central processing unit*), and can be obtained using the MATLAB command `cputime`. The total elapsed time between the *input* and *output* phases can be obtained by the command `etime`.

cputime
etime

**Example 1.4** In order to compute the time needed for a matrix-vector multiplication we set up the following program:

```
>> n=10000; step=100;
>> A=rand(n,n);
>> v=rand(n,1);
>> T=[ ];
>> sizeA=[ ];
>> for k = 500:step:n
    AA = A(1:k,1:k);
    vv = v(1:k)';
    t = cputime;
    b = AA*vv;
    tt = cputime - t;
    T = [T, tt];
    sizeA = [sizeA,k];
 end
```

a:step:b
rand

The instruction `a:step:b` appearing in the `for` cycle generates all numbers having the form `a+step*k` where `k` is an integer ranging from `0` to the largest value `kmax` for which `a+step*kmax` is not greater than `b` (in the case at hand, `a=500`, `b=10000` and `step=100`). The command `rand(n,m)` defines an `n`×`m` matrix of random entries. Finally, `T` is the vector whose components contain the CPU time needed to carry out every single matrix-vector product, whereas `cputime` returns the CPU time in seconds that has been used by the MATLAB process since MATLAB started. The time necessary to execute a single program is therefore the difference between the actual CPU time and the one computed before the execution of the current program which is stored in the variable `t`. Figure 1.9, which is obtained by the command `plot(sizeA,T,'o')`, shows that the CPU time grows like the square of the matrix order `n`. ■

## 1.7 The MATLAB language

After the introductory remarks of the previous section, we are now ready to work in either the MATLAB or Octave environments. As said above, from now on MATLAB should be understood as the subset of commands which are common to both MATLAB and Octave.

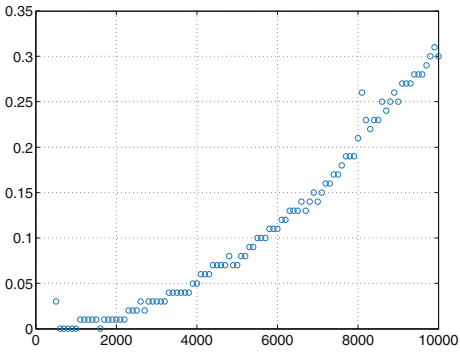


**Fig. 1.9.** Matrix-vector product: the CPU time (in seconds) versus the dimension $n$ of the matrix (on an Intel® Core™2 Duo, 2.53 GHz processor)

After pressing the *enter* key (or else *return*), all what is written after the *prompt* will be interpreted.[1] Precisely, MATLAB will first check whether what is written corresponds either to variables which have already been defined or to the name of one of the programs or commands defined in MATLAB. Should all those checks fail, MATLAB returns an error warning. Otherwise, the command is executed and an *output* will possibly be displayed. In all cases, the system eventually returns the *prompt* to acknowledge that it is ready for a new command. To close a MATLAB session one should write the command `quit` (or else `exit`) and press the *enter* key. From now it will be understood that to execute a program or a command one has to press the *enter* key. Moreover, the terms program, function or command will be used in an equivalent manner. When our command coincides with one of the elementary structures characterizing MATLAB (e.g. a number or a string of characters that are put between apices) they are immediately returned in *output* in the *default* variable `ans` (abbreviation of *answer*). Here is an example:

`quit` `exit` `ans`

```
>>  'home'
 ans =
    home
```

If we now write a different string (or number), `ans` will assume this new value.

We can turn off the automatic display of the *output* by writing a semicolon after the string. Thus if we write `'home';` MATLAB will simply return the *prompt* (yet assigning the value `'home'` to the variable `ans`).

More generally, the command `=` allows the assignment of a value (or `=`

[1] Thus a MATLAB program does not necessarily have to be compiled as other languages do, e.g. Fortran or C.

a string of characters) to a given variable. For instance, to assign the string `'Welcome to Milan'` to the variable `a` we can write

```
>> a='Welcome to Milan';
```

Thus there is no need to declare the *type* of a variable, MATLAB will do it automatically and dynamically. For instance, should we write `a=5`, the variable `a` will now contain a number and no longer a string of characters. This flexibility is not cost-free. If we set a variable named `quit` equal to the number `5` we are inhibiting the use of the MATLAB command `quit`. We should therefore try to avoid using variables having the name of MATLAB commands. However, by the command `clear` followed by the name of a variable (e.g. `quit`), it is possible to cancel this assignment and restore the original meaning of the command `quit`.

clear

By the command `save` all the session variables (that are stored in the so-called *base workspace*) are saved in the binary file `matlab.mat`. Similarly, the command `load` restores in the current session all variables stored in `matlab.mat`. A file name can be specified after `save` or `load`. One can also save only selected variables, say `v1`, `v2` and `v3`, in a given file named, e.g., `area.mat`, using the command `save area v1 v2 v3`.

save

load

By the command `help` one can see the whole family of commands and pre-defined variables, including the so-called *toolboxes* which are sets of specialized commands. Among them let us recall those which define the elementary functions such as sine (`sin(a)`), cosine (`cos(a)`), square root (`sqrt(a)`), exponential (`exp(a)`).

help

sin cos sqrt exp

There are special characters that cannot appear in the name of a variable or in a command, for instance the algebraic operators (`+`, `-`, `*` and `/`), the logical operators *and* (`&`), *or* (`|`), *not* (`~`), the relational operators *greater than* (`>`), *greater than or equal to* (`>=`), *less than* (`<`), *less than or equal to* (`<=`), *equal to* (`==`). Finally, a name can never begin with a digit, and it cannot contain a bracket or any punctuation mark.

+ - * / & | ~ > >= < <= ==

### 1.7.1 MATLAB statements

A special programming language, the MATLAB language, is also available enabling the users to write new programs. Although its knowledge is not required for understanding how to use the several programs which we will introduce throughout this book, it may provide the reader with the capability of modifying them as well as producing new ones.

The MATLAB language features standard statements, such as conditionals and loops.

The *if-elseif-else* conditional has the following general form:

```
if <condition 1>
   <statement 1.1>
   <statement 1.2>
   ...
elseif <condition 2>
```

```
   <statement 2.1>
   <statement 2.2>
   ...
...
else
   <statement n.1>
   <statement n.2>
   ...
end
```

where `<condition 1>`, `<condition 2>`, ... represent MATLAB sets of logical expressions, with values 0 or 1 (false or true) and the entire construction allows the execution of that statement corresponding to the condition taking value equal to 1. Should all conditions be false, the execution of `<statement n.1>`, `<statement n.2>`, `...` will take place. In fact, if the value of `<condition k>` is zero, the statements `<statement k.1>`, `<statement k.2>`, `...` are not executed and the control moves on.

For instance, to compute the roots of a quadratic polynomial $ax^2 + bx + c$ one can use the following instructions (the command `disp(.)` simply displays what is written between brackets):

disp

```
>> if  a  ~= 0
     sq = sqrt(b*b - 4*a*c);
     x(1) = 0.5*(-b + sq)/a;
     x(2) = 0.5*(-b - sq)/a;
   elseif  b  ~= 0
     x(1) = -c/b;
   elseif  c  ~= 0
     disp(' Impossible  equation');
   else
     disp(' The  given  equation  is  an  identity');
   end
```

(1.13)

Note that MATLAB does not execute the entire construction until the statement `end` is typed.

MATLAB allows two types of loops, a `for`-*loop* (comparable to a Fortran *do-loop* or a C *for-loop*) and a `while`-*loop*. A for-loop repeats the statements in the loop as the loop index takes on the values in a given row vector. For instance, to compute the first six terms of the Fibonacci sequence $f_i = f_{i-1} + f_{i-2}$, for $i \geq 3$, with $f_1 = 0$ and $f_2 = 1$, one can use the following instructions:

for

while

```
>> f(1) = 0; f(2) = 1;
>> for i = [3 4 5 6]
     f(i) = f(i-1) + f(i-2);
 end
```

Note that a semicolon can be used to separate several MATLAB instructions typed on the same line. Also, note that we can replace the second instruction by the equivalent `>> for i = 3:6`. The while-loop repeats

as long as the given `condition` is true. For instance, the following set of instructions can be used as an alternative to the previous set:

```
>> f(1) = 0; f(2) = 1; k = 3;
>> while k <= 6
    f(k) = f(k-1) + f(k-2); k = k + 1;
 end
```

Other statements of perhaps less frequent use exist, such as `switch`, `case`, `otherwise`. The interested reader can have access to their meaning by the `help` command.

### 1.7.2 Programming in MATLAB

Let us now explain briefly how to write MATLAB programs. A new program must be put in a file with a given name with extension `m`, which is called *m-file*. They must be located in one of the directories in which MATLAB automatically searches for m-files; their list can be obtained by the command `path` (see `help path` to learn how to add a directory to this list). The first directory scanned by MATLAB is the current working directory.

`path`

It is important at this level to distinguish between *scripts* and *functions*. A script is simply a collection of MATLAB commands in an *m-file* and can be used interactively. For instance, the set of instructions (1.13) can give rise to a script (which we could name `equation`) by copying it in the file `equation.m`. To launch it, one can simply write the instruction `equation` after the MATLAB prompt `>>`. We report two examples below:

```
>> a = 1; b = 1; c = 1;
>> equation
>> x

 x =
   -0.5000 + 0.8660i   -0.5000 - 0.8660i

>> a = 0; b = 1; c = 1;
>> equation
>> x

 x =
     -1
```

Since we have no input/output interface, all variables used in a *script* are also the variables of the working session and are therefore cleared only upon an explicit command (`clear`). This is not at all satisfactory when one intends to write complex programs involving many temporary variables and comparatively fewer input and output variables, which are the only ones that can be effectively saved once the execution of the program is terminated. Much more flexible than scripts are *functions*.

A *function* is still defined in a m-file, e.g. `name.m`, but it has a well defined input/output interface that is introduced by the command `function`

```
function [out1,...,outn]=name(in1,...,inm)
```

where `out1,...,outn` are the output variables and `in1,...,inm` are the input variables.

The following file, called `det23.m`, defines a new function called `det23` which computes, according to the formulae given in Section 1.4, the determinant of a matrix whose dimension could be either 2 or 3:

```
function det=det23(A)
%DET23 computes the determinant of a square matrix
% of dimension 2 or 3
[n,m]=size(A);
if n==m
  if n==2
    det = A(1,1)*A(2,2)-A(2,1)*A(1,2);
  elseif n == 3
    det = A(1,1)*det23(A([2,3],[2,3]))-...
          A(1,2)*det23(A([2,3],[1,3]))+...
          A(1,3)*det23(A([2,3],[1,2]));
  else
    disp(' Only 2x2 or 3x3 matrices ');
  end
else
  disp(' Only square matrices ');
end
return
```

Notice the use of the continuation characters `...` meaning that the instruction is continuing on the next line and the character `%` to begin comments. The instruction `A([i,j],[k,l])` allows the construction of a $2 \times 2$ matrix whose elements are the elements of the original matrix A lying at the intersections of the `i`-th and `j`-th rows with the `k`-th and `l`-th columns.

When a function is invoked, MATLAB creates a local workspace (the *function's workspace*). The commands in the function cannot refer to variables from the base (interactive) workspace unless they are passed as input.[2] In particular, variables used in a function are erased when the execution terminates, unless they are returned as output parameters.

Functions usually terminate when the end of the function is reached, however a `return` statement can be used to force an early return (upon the fulfillment of a certain condition).

For instance, in order to approximate the golden section number $\alpha = 1.6180339887\ldots$, which is the limit for $k \to \infty$ of the quotient of two consecutive Fibonacci numbers $f_k/f_{k-1}$, by iterating until the difference

[2] A third type of workspace, the so called global workspace, is available and is used to store global variables. These variables can be used inside a function even if they are not among the input parameters.

between two consecutive ratios is less than $10^{-4}$, we can construct the following function:

```
function [golden,k]=fibonacci0
% FIBONACCI0: Golden section number approximation
f(1) = 0; f(2) = 1; goldenold = 0;
kmax = 100; tol = 1.e-04;
for k = 3:kmax
f(k) = f(k-1) + f(k-2); golden = f(k)/f(k-1);
if abs(golden - goldenold) < tol
return
end
goldenold = golden;
end
return
```

Its execution is interrupted either after `kmax=100` iterations or when the absolute value of the difference between two consecutive iterates is smaller than `tol=1.e-04`. Then, we can write

```
>> [alpha,niter]=fibonacci0
  alpha =
     1.61805555555556
  niter =
     14
```

After 14 iterations the function has returned an approximate value which shares with $\alpha$ the first 5 significant digits.

The number of input and output parameters of a MATLAB function can vary. For instance, we could modify the Fibonacci function as follows:

```
function [golden,k]=fibonacci1(tol,kmax)
% FIBONACCI1: Golden section number approximation
%   Both tolerance and maximum number of iterations
%   can be assigned in input
if nargin == 0
  kmax = 100; tol = 1.e-04; % default values
elseif nargin == 1
  kmax = 100; % default value of kmax
end
f(1) = 0; f(2) = 1; goldenold = 0;
for k = 3:kmax
   f(k) = f(k-1) + f(k-2);
   golden = f(k)/f(k-1);
   if abs(golden - goldenold) < tol
     return
   end
   goldenold = golden;
end
return
```

nargin
nargout

The `nargin` function counts the number of input parameters (in a similar way the `nargout` function counts the number of output parameters). In the new version of the `fibonacci` function we can prescribe a specific tolerance `tol` and the maximum number of inner iterations allowed (`kmax`). When this information is missing the function must provide default values (in our case, `tol = 1.e-04` and `kmax = 100`). A possible use of it is as follows:

```
>> [alpha,niter]=fibonacci1(1.e-6,200)
 alpha =
    1.61803381340013
 niter =
     19
```

Note that using a stricter tolerance we have obtained a new approximate value that shares with $\alpha$ as many as 8 significant digits.
The `nargin` function can be used externally to a given function to obtain the number of input parameters. Here is an example:

```
>> nargin('fibonacci1')
 ans =
      2
```

After this quick introduction, our suggestion is to explore MATLAB using the command *help*, and get acquainted with the implementation of various algorithms by the programs described throughout this book. For instance, by typing `help for` we get not only a complete description on the command `for` but also an indication on instructions similar to `for`, such as `if`, `while`, `switch`, `break` and `end`. By invoking their *help* we can progressively improve our knowledge of MATLAB.

### 1.7.3 Examples of differences between MATLAB and Octave languages

As already mentioned, what has been written in the previous section about the MATLAB language applies to both MATLAB and Octave environments without changes. However, some differences exist for the language itself. So programs written in Octave may not run in MATLAB and viceversa. For example, Octave supports strings with single and double quotes

```
octave:1> a="Welcome to Milan"
a = Welcome to Milan
octave:2> a='Welcome to Milan'
a = Welcome to Milan
```

whereas MATLAB supports only single quotes, double quotes will result in parsing errors.

Here we provide a list of few other incompatibilities between the two languages:

- MATLAB does not allow a blank before the transpose operator. For instance, `[0 1]'` works in MATLAB, but `[0 1] '` does not. Octave properly parses both cases;
- MATLAB always requires ...,

  ```
  rand (1, ...
        2)
  ```

  while both

```
rand (1,
      2)
```

and

```
rand (1, \
      2)
```

work in Octave in addition to ...;
- for exponentiation, Octave can use ^ or **; MATLAB requires ^;
- for ends, Octave can use `end` but also `endif`, `endfor`, ...; MATLAB requires `end`.

See Exercises 1.9-1.14.

## 1.8 What we haven't told you

A systematic discussion on floating-point numbers can be found in [Übe97], [Hig02] and in [QSS07].

For matters concerning the issue of complexity, we refer, e.g., to [Pan92].

For a more systematic introduction to MATLAB the interested reader can refer to the MATLAB manual [HH05] as well as to specific books such as [HLR06], [Pra06], [EKM05], [Pal08] or [MH03].

For Octave we recommend the manual book mentioned at the beginning of this chapter.

## 1.9 Exercises

**Exercise 1.1** How many numbers belong to the set $\mathbb{F}(2, 2, -2, 2)$? What is the value of $\epsilon_M$ for such set?

**Exercise 1.2** Show that the set $\mathbb{F}(\beta, t, L, U)$ contains precisely $2(\beta - 1)\beta^{t-1}$ $(U - L + 1)$ elements.

**Exercise 1.3** Prove that $i^i$ is a real number, then check this result using MATLAB.

**Exercise 1.4** Write the MATLAB instructions to build an upper (respectively, lower) triangular matrix of dimension 10 having 2 on the main diagonal and $-3$ on the second upper (respectively, lower) diagonal.

**Exercise 1.5** Write the MATLAB instructions which allow the interchange of the third and seventh row of the matrices built up in Exercise 1.4, and then the instructions allowing the interchange between the fourth and eighth column.

**Exercise 1.6** Verify whether the following vectors in $\mathbb{R}^4$ are linearly independent:

$$\mathbf{v}_1 = [0\ 1\ 0\ 1],\ \mathbf{v}_2 = [1\ 2\ 3\ 4],\ \mathbf{v}_3 = [1\ 0\ 1\ 0],\ \mathbf{v}_4 = [0\ 0\ 1\ 1].$$

**Exercise 1.7** Write the following functions and compute their first and second derivatives, as well as their primitives, using the symbolic toolbox of MATLAB:

$$f(x) = \sqrt{x^2 + 1}, \qquad g(x) = \sin(x^3) + \cosh(x).$$

**Exercise 1.8** For any given vector `v` of dimension $n$, using the command `c=poly(v)` one can construct the $n+1$ coefficients of the polynomial $p(x) = \sum_{k=1}^{n+1} \mathtt{c}(k) x^{n+1-k}$ which is equal to $\Pi_{k=1}^{n}(x - \mathtt{v}(k))$. In exact arithmetics, one should find that `v = roots(poly(v))`. However, this cannot occur due to roundoff errors, as one can check by using the command `roots(poly([1:n]))`, where `n` ranges from 2 to 25. poly

**Exercise 1.9** Write a program to compute the following sequence:

$$I_0 = \frac{1}{e}(e-1),$$
$$I_{n+1} = 1 - (n+1)I_n, \text{ for } n = 0, 1, \ldots.$$

Compare the numerical result with the exact limit $I_n \to 0$ for $n \to \infty$.

**Exercise 1.10** Explain the behavior of the sequence (1.4) when computed in MATLAB.

**Exercise 1.11** Consider the following algorithm to compute $\pi$. Generate $n$ couples $\{(x_k, y_k)\}$ of random numbers in the interval $[0, 1]$, then compute the number $m$ of those lying inside the first quarter of the unit circle. Obviously, $\pi$ turns out to be the limit of the sequence $\pi_n = 4m/n$. Write a MATLAB program to compute this sequence and check the error for increasing values of $n$.

**Exercise 1.12** Since $\pi$ is the sum of the series

$$\pi = \sum_{n=0}^{\infty} 16^{-n} \left( \frac{4}{8n+1} - \frac{2}{8n+4} - \frac{1}{8n+5} - \frac{1}{8n+6} \right).$$

we can compute an approximation of $\pi$ by summing up to the $n$-th term, for a sufficiently large $n$. Write a MATLAB *function* to compute finite sums of the above series. How large should $n$ be in order to obtain an approximation of $\pi$ at least as accurate as the one stored in the variable $\pi$?

**Exercise 1.13** Write a program for the computation of the binomial coefficient $\binom{n}{k} = n!/(k!(n-k)!)$, where $n$ and $k$ are two natural numbers with $k \le n$.

**Exercise 1.14** Write a recursive MATLAB *function* that computes the $n$-th element $f_n$ of the Fibonacci sequence. Noting that

$$\begin{bmatrix} f_i \\ f_{i-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} f_{i-1} \\ f_{i-2} \end{bmatrix} \tag{1.14}$$

write another *function* that computes $f_n$ based on this new recursive form. Finally, compute the related CPU-time.