TEXTS IN COMPUTATIONAL SCIENCE AND ENGINEERING 2

Alfio Quarteroni
Fausto Saleri · Paola Gervasio

Scientific Computing with MATLAB and Octave

Fourth Edition

Springer

Texts in Computational
Science and Engineering 2

Editors

Timothy J. Barth
Michael Griebel
David E. Keyes
Risto M. Nieminen
Dirk Roose
Tamar Schlick

Alfio Quarteroni • Fausto Saleri •
Paola Gervasio

Scientific Computing with MATLAB and Octave

Fourth Edition

Springer

Alfio Quarteroni
MATHICSE-CMCS
Ecole Polytechnique Fédérale
de Lausanne
Lausanne
Switzerland

Fausto Saleri (1965-2007)
MOX - Politecnico di Milano
Milano
Italy

Paola Gervasio
DICATAM
Università degli Studi di Brescia
Brescia
Italy

ISSN 1611-0994
ISBN 978-3-642-45366-3 ISBN 978-3-642-45367-0 (eBook)
DOI 10.1007/978-3-642-45367-0
Springer Heidelberg New York Dordrecht London

Library of Congress Control Number: 2014932425

Mathematics Subject Classification (2010): 65-01, 68U01, 68N15

The picture on the cover shows a re-entrant electrical wave on a slab of homogeneous excitable medium, exhibiting spiral turbulence and spatiotemporal chaos. Computation by Ricardo Ruiz-Baier, IST, University of Lausanne, CH.

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

To the memory of
Fausto Saleri

Preface

Preface to the First Edition

This textbook is an introduction to Scientific Computing. We will illustrate several numerical methods for the computer solution of certain classes of mathematical problems that cannot be faced by paper and pencil. We will show how to compute the zeros or the integrals of continuous functions, solve linear systems, approximate functions by polynomials and construct accurate approximations for the solution of differential equations.

With this aim, in Chapter 1 we will illustrate the rules of the game that computers adopt when storing and operating with real and complex numbers, vectors and matrices.

In order to make our presentation concrete and appealing we will adopt the programming environment MATLAB® [1] as a faithful companion. We will gradually discover its principal commands, statements and constructs. We will show how to execute all the algorithms that we introduce throughout the book. This will enable us to furnish an immediate quantitative assessment of their theoretical properties such as stability, accuracy and complexity. We will solve several problems that will be raised through exercises and examples, often stemming from specific applications.

Several graphical devices will be adopted in order to render the reading more pleasant. We will report in the margin the MATLAB command along side the line where that command is being introduced for the first time. The symbol will be used to indicate the presence of exercises, the symbol to indicate the presence of a MATLAB program, while

[1] MATLAB is a trademark of TheMathWorks Inc., 24 Prime Park Way, Natick, MA 01760, Tel: 001+508-647-7000, Fax: 001+508-647-7001.

the symbol will be used when we want to attract the attention of the reader on a critical or surprising behavior of an algorithm or a procedure. The mathematical formulae of special relevance are put within a frame. Finally, the symbol indicates the presence of a display panel summarizing concepts and conclusions which have just been reported and drawn.

At the end of each chapter a specific section is devoted to mentioning those subjects which have not been addressed and indicate the bibliographical references for a more comprehensive treatment of the material that we have carried out.

Quite often we will refer to the textbook [QSS07] where many issues faced in this book are treated at a deeper level, and where theoretical results are proven. For a more thorough description of MATLAB we refer to [HH05]. All the programs introduced in this text can be downloaded from the web address

`mox.polimi.it/qs`

No special prerequisite is demanded of the reader, with the exception of an elementary course of Calculus.

However, in the course of the first chapter, we recall the principal results of Calculus and Geometry that will be used extensively throughout this text. The less elementary subjects, those which are not so necessary for an introductory educational path, are highlighted by the special symbol .

We express our thanks to Thanh-Ha Le Thi from Springer-Verlag Heidelberg, and to Francesca Bonadei and Marina Forlizzi from Springer-Italia for their friendly collaboration throughout this project. We gratefully thank Prof. Eastham of Cardiff University for editing the language of the whole manuscript and stimulating us to clarify many points of our text.

Milano and Lausanne *Alfio Quarteroni*
May 2003 *Fausto Saleri*

Preface to the Second Edition

In this second edition we have enriched all the Chapters by introducing several new problems. Moreover, we have added new methods for the numerical solution of linear and nonlinear systems, the eigenvalue computation and the solution of initial-value problems. Another relevant improvement is that we also use the Octave programming environment. Octave is a reimplementation of part of MATLAB which includes many numerical facilities of MATLAB and is freely distributed under the GNU General Public License.

Throughout 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. We have striven to ensure a seamless usage of our codes and programs under both MATLAB and Octave. In the few cases where this does not apply, we shall write a short explanation notice at the end of each corresponding section.

For this second edition we would like to thank Paola Causin for having proposed several problems, Christophe Prud´homme, John W. Eaton and David Bateman for their help with Octave, and Silvia Quarteroni for the translation of the new sections. Finally, we kindly acknowledge the support of the Poseidon project of the Ecole Polytechnique Fédérale de Lausanne.

Lausanne and Milano *Alfio Quarteroni*
May 2006 *Fausto Saleri*

Preface to the Third Edition

This third edition features a complete revisitation of the whole book, many improvements in style and content to all the chapters, as well as a substantial new development of those chapters devoted to the numerical approximation of boundary-value problems and initial-boundary-value problems. We remind the reader that all the programs introduced in this text can be downloaded from the web address

`mox.polimi.it/qs`

Lausanne, Milano and Brescia *Alfio Quarteroni*
March 2010 *Paola Gervasio*

Preface to the Fourth Edition

The fourth edition features the addition of a new chapter on numerical optimization of both univariate and multivariate functions in which several methods are presented, discussed and analyzed.

For unconstrained minimization, we consider derivative free methods, descent (or line search) methods, and trust region methods.

For constrained minimization we restrict our discussion to penalization methods and augmented Lagrangian methods.

As for the other chapters of this book, also this new chapter is supported by examples, exercises and programs written in both MATLAB and Octave environments.

The addition of this chapter made it necessary a renumbering of several other chapters with respect to the previous editions. Moreover, new sections have been added in some other chapters.

Finally we remind the reader that all programs presented in this book can be downloaded from the web address

`http://mox.polimi.it/qs`

Lausanne, Milano and Brescia
December 2013

Alfio Quarteroni
Paola Gervasio

Contents

Index of MATLAB and Octave programs

All the programs introduced in this text can be downloaded from
`http://mox.polimi.it/qs`

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 intepreter 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 [Att11] for a description of the MATLAB language and to [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 1984-2013 The MathWorks, Inc.
             R2013b (8.2.0.701) 64-bit (glnxa64)
                     August 13, 2013

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 3.6.4
Copyright (C) 2013 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/get-involved.html
Read http://www.octave.org/bugs.html to learn how to
submit bug reports.
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 normalized *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 available 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     "    1.42857142857143e - 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

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

where s is either 0 or 1, β (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 β, 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`). Floating point numbers of $\mathbb{F}(2, 53, -1021, 1024)$ are stored in registers of 8 Bytes, more precisely the sign s is stored in 1 bit, the exponent e in 11 bits, and the mantissa m in 52 bits. Note that, although we have 52 bits for m, we can count $t = 53$ digits when $\beta = 2$. As a matter of fact, since the first digit a_1 of every floating point number must be different from 0, when $\beta = 2$ it is worthless to store it as it must necessarily be 1. The digits $a_2, \dots, a_{53}$ are therefore stored in the 52 bits associated with m.

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

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

where $\epsilon_M = \beta^{1-t}$, which is called *machine epsilon*, provides the distance between 1 and its closest floating-point number greater than 1. Note that ϵ_M depends on β and t. For instance, in MATLAB ϵ_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

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

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

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

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 ϵ_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

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

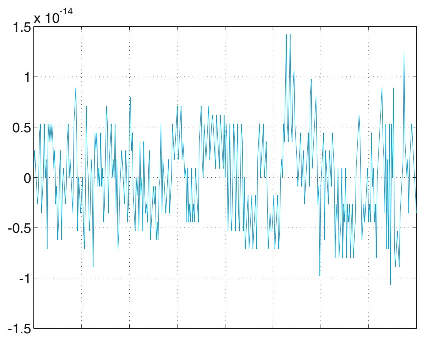


Figure 1.1. Oscillatory behavior of the function (1.3) caused by cancellation errors

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 $((1+x)-1)/x$ (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 ∞/∞. 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,

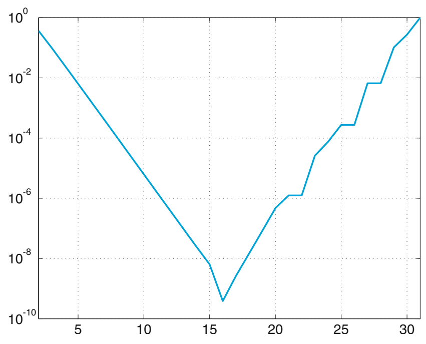


Figure 1.2. Relative error $|\pi - z_n|/\pi$ versus n

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

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

which converges to π when n tends to infinity. (This sequence is a revised form of the better known *formula of François Viète* (french mathematician of the XVI century) for the approximation of π [Bec71].)

When MATLAB is used to compute z_n, the relative error found between π 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 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*

complex

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

$\rho = \sqrt{x^2+y^2}$ is the modulus of the complex number (it can be obtained by setting `abs(z)`) while θ is its argument, that is the angle between the

abs

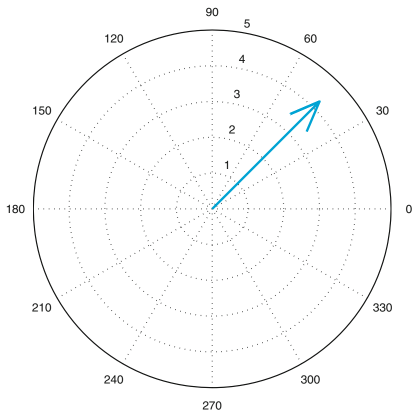


Figure 1.3. Output of the MATLAB command `compass`

x axis and the straight line issuing from the origin and passing from the point of coordinate x, y in the complex plane. θ can be found by typing `angle(z)`. The representation (1.5) is therefore:

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

angle

The graphical polar representation of one or more complex numbers 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

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

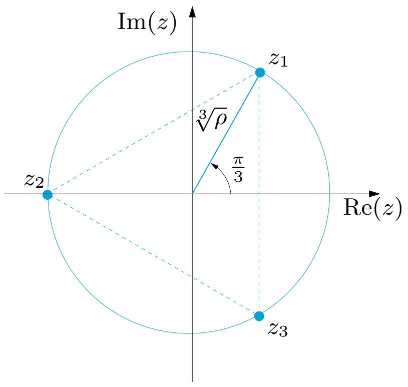


Figure 1.4. Representation in the complex plane of the three complex cube roots of the real number -5

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)

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

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

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

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 $i = 1,\dots,m$, $j = 1,\dots,n$, represented by the following table:

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

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 2×3 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 λ 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 ==> plus
 Matrix dimensions must agree.
>> A*B
 ??? Error using ==> mtimes
 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 A^{-1}, which satisfies the matrix relation $\mathrm{AA}^{-1} = \mathrm{A}^{-1}\mathrm{A} = \mathrm{I}$. We can obtain A^{-1} through the

`inv` `det`

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 A_{ij} is the matrix obtained by eliminating the ith row and jth 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

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

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

To invert a 2×2 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 mth 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 $A \in \mathbb{R}^{m\times n}$ its transpose $A^T \in \mathbb{R}^{n\times m}$ is the matrix obtained by interchanging rows and columns of A. When $n = m$ and $A = A^T$ the matrix A is called *symmetric*. Finally, `A'` denotes the transpose of `A` if `A` is real, or its conjugate transpose (that is, A^H) if `A` is complex. A square complex matrix that coincides with its conjugate transpose 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)

octave:5> A*B

error: operator *: nonconformant arguments (op1 is
x3, op2 is 2x3)
```

■

1.4.1 Vectors

Vectors will be indicated in boldface; precisely, $\mathbf{v}$ will denote a column vector whose ith 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 ith 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.

`dot`

`v'`

The length (or modulus) of a vector $\mathbf{v}$ is given by

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

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 α 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`

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

`quiver3`

`.*  ./ .`

```
>> 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 ith component is equal to $x_i y_i$. Note that the product `y*x` is not well-defined.

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 λ if

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

The complex number λ 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.

Let us consider a real function, for example $f(x) = 1/(1 + x^2)$; we are going to show the MATLAB instructions to define it, evaluate it at a point (or on a set of points), and plot it.
The simplest way to define a mathematical function consists in using *anonymous function* and *function handle* `@` as follows:

@

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

and we can evaluate f at $x = 3$ by the instruction

```
>> y=fun(3)
 y =
    0.1000
```

An *anonymous function* is a function that is not stored in a program file, but is associated with a variable whose data type is a *function handle*, that is a MATLAB standard variable that provides a means of calling a function. Function handles can be passed in calls to other MATLAB functions.

The common syntax to create a handle associated with an anonymous function reads

```
>> fun=@(arg1, arg2,...,argn)expr
```

where `fun` is the function handle, `arg1, arg2,...,argn` are the independent variables of the anonymous function, while `expr` contains the expression of the anonymous function we want to define; it could be included between round or square brackets.

Some parameters can be used inside `expr`, even if they do not appear in the list of variables `(arg1, arg2,...,argn)`. If this is the case, such parameters must be set before the function definition. For example, to evaluate $f(x) = a/(1+x^2)$ at $x = 2$ and with $a = 3$, we write:

```
>> a=3; fun=@(x) a/(1+x^2); y=fun(2)
```

and the result is

```
 y =
    0.6000
```

To modify the value of the parameter a, e.g. to set $a = 8$, we have to define again the function handle `fun`, otherwise MATLAB holds on the value `a=3` in `fun`. Then we must write

```
>> a=8; fun=@(x) a/(1+x^2); y=fun(2)
 y =
    1.6000
```

The command `fplot(fun,lims)` plots the graph of the function associated with the function handle `fun` 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

`fplot`

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

or, more directly, by invoking the anonymous function without function handle,

```
>> fplot(@(x) 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`(≥ 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`

An alternative way for defining mathematical functions consists in writing MATLAB functions, also called *user-defined functions*. (See Sect. 1.7.2.) For instance, we can write the following instructions

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

and save them into the file `fun.m`. It is suggested (although not mandatory) that the filename coincides with the name written in the first row of the file itself. As a matter of fact, should we save the above three rows in a file with a different name, e.g. `funct.m`, MATLAB will recognise the user-defined function `funct`, but it will not be able to locate `fun`.

To plot on the interval $[-\pi, \pi]$ the function defined in `fun.m`, we can use one of the following commands:

```
>> fplot(@fun,[-pi,pi])
```

or

```
>> fplot('fun',[-pi,pi])
```

(The special character `@` generates the function handle associated with function `fun`.)

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.

Remark 1.2 Function handles can also be associated with vector functions. In that case, the common syntax to define arrays is used, e.g. spaces or commas are used to separate different elements of a row, while semicolons to split rows. For example, to define the vector function $\mathbf{g} : \mathbb{R}^2 \to \mathbb{R}^2$, with $\mathbf{g}(x, y) = [e^x \sin(y),\ x^2 - y]^t$ we can use the command

```
>> g=@(x,y) [exp(x).*sin(y); x.^2-y]
```

In order to prevent wrong function definitions, we warn the reader that blanks should be avoided when unnecessary. For instance, by writing $f(x) = 2x - sin(x)$ introducing a blank between `2*x` and `-sin(x)`

```
>> f=@(x) [2*x  -sin(x)],
```

the evaluation

```
>> y=f(pi/2)
```

provides the row arrow

```
y=
    3.1416    -1.0000
```

instead of the scalar value `y=3.1416`. Actually, the space between `2*x` and `-sin(x)` is interpreted as separation character and this means that the above instruction defines the vector function $\mathbf{f} : \mathbb{R} \to \mathbb{R}^2$ $\mathbf{f}(x) = [2x, \sin(x)]$ instead of $f(x) = 2x - sin(x)$. ■

1.5.1 The zeros

We recall that if $f(\alpha) = 0$, α 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, α_+ and α_-, when $n = 2$ (this time p_2 represents a parabola) $\alpha_\pm = (-a_1 \pm \sqrt{a_1^2 - 4a_0a_2})/(2a_2)$.

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,

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

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.

If `fun` is defined by a user-defined function, we can choose one between these two calls:

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

or

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

Octave 1.2 When the command `fzero(fun,x0)` is used being `x0` a scalar value, Octave suggests to use the *function* `fsolve`. ■

1.5.2 Polynomials

`polyval` Polynomials are very special functions for whose treatment special MATLAB functions are available. 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 $k = 0, \ldots, 8$, 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 anonymous functions and function handles. 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 roundoff errors (see Section 2.8.2).

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

Table 1.1. MATLAB commands for polynomial operations

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 $\int_0^x p(t)\ dt$

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
>> [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, `p`$_1$ and `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 1.1.

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 $f \in C^0([a,b])$ and it is differentiable in (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:

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

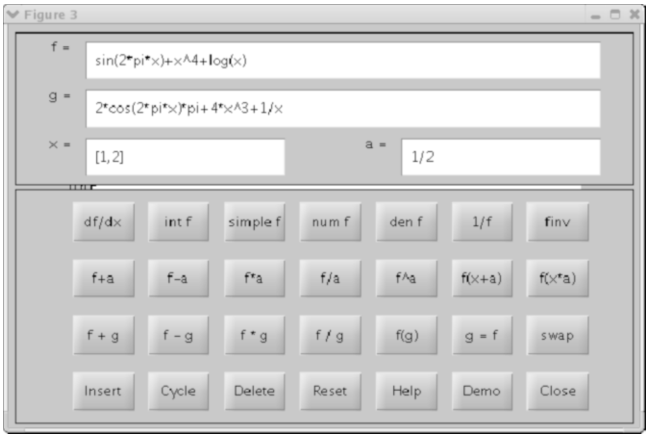


Figure 3

f = sin(2*pi*x)+x^4+log(x)

g = 2*cos(2*pi*x)*pi+4*x^3+1/x

x = [1,2]

a = 1/2

Figure 1.5. Graphical interface of the command `funtool`

diff int taylor

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.

syms

First of all, the variable `x` must be declared *symbolic* by the command `syms x`. This will allow its algebraic manipulation without specifying its value.

Therefore, having defined the expression `f` of the function on which we intend to operate, `diff(f,n)` provides its derivative of order `n`, `int(f)` its indefinite integral, and `taylor(f,'Order',n+1)` the associated Taylor polynomial of degree `n` in a neighborhood of $x_0 = 0$.

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

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

simple

funtool

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.

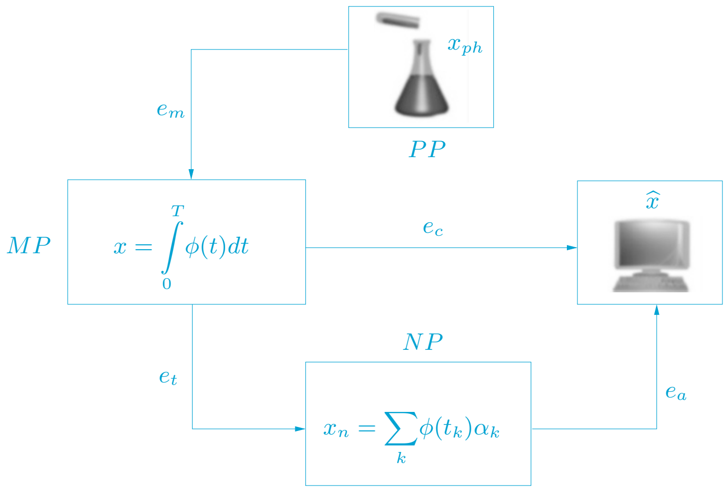


Figure 1.6. Types of errors in a computational process

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.

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 $\bar{x}$. ■

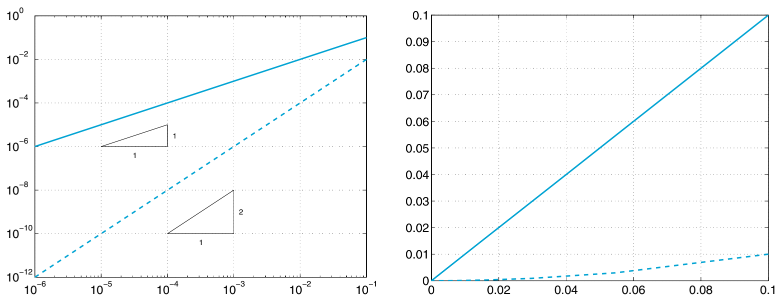


Figure 1.7. Plot of the same data in log-log scale *(left)* and in linear-linear scale *(right)*

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, $p = 2$ 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, 2.6 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.

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

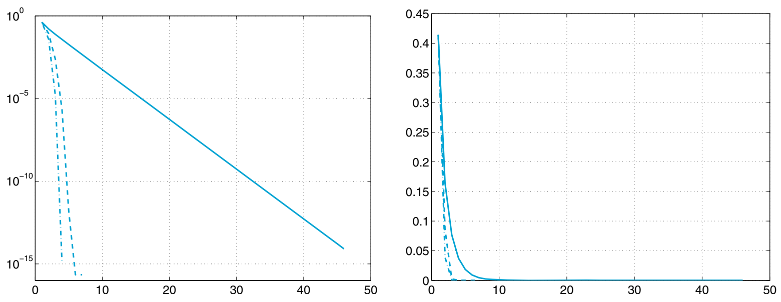


Figure 1.8. Errors e_n^x *(solid line)*, e_n^y *(dashed line)* and e_n^z *(dashed-dotted line)* in semi-logarithmic scale *(left)* and linear-linear scale *(right)*

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 computer nowadays (1st of the top500 supercomputer list as of November 2013) reaches as many as 33 Peta-flops and is the Tianhe-2 (MilkyWay-2) Cluster, Intel Xeon 2.200GHz, of National University of Defense Technology, China.

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 jth 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

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. ■

rand

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,

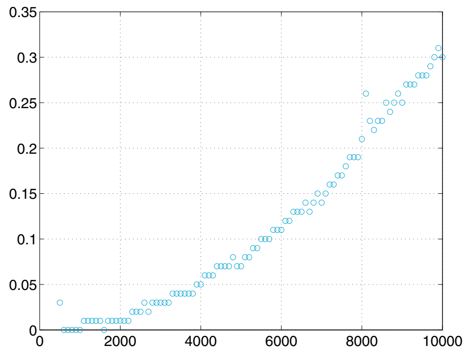


Figure 1.9. Matrix-vector product: the CPU time (in seconds) versus the dimension n of the matrix (on an Intel® Core™2 Duo, 2.53 GHz processor)

from now on MATLAB should be understood as the subset of commands which are common to both MATLAB and Octave.

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`).

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

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

`save` 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`.

`load`

`help` 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)`).

`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 (or boolean) 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;                                      (1.13)
   elseif  c  ~= 0
     disp(' Impossible  equation');
   else
     disp(' The  given  equation  is  an  identity');
   end
```

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

Logical expressions that appear inside conditional statements can be obtained by combining elementary logical expressions using boolean operators `&`, `|`, `&&`, and `||`. The two latter operators `&&` and `||` implement the *short-circuiting* capability of the corresponding *element-by-element* ones `&` and `|`. As a matter of fact, element-by-element boolean operators are often sufficient for performing most logical operations. However, it is sometimes desirable to stop evaluating a logical expression as soon as the overall truth value can be determined.

More precisely, the second operand of the logical expression `(expr1 && expr2)` (or `(expr1 || expr2)`) is evaluated only if the result is not fully determined by the first one. For instance, let us consider the following expression

```
>> (nit <= nmax) & (err > tol)
```

MATLAB first computes the value of both operands and then the result of the boolean operation *and*. Instead, by the instruction

```
>> (nit <= nmax) && (err > tol)
```

MATLAB computes the value of the first operand and, only if it is equal to 1 (that is true), MATLAB computes the value of the second one.

for
while

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:

```
>> 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 *user-defined 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 *user-defined function* (in brief, *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` as follows

`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 2×2 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.

return 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 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 α the first 5 significant digits.

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

[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.

```
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
```

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:

nargin
nargout

```
>> [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 α 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 not equal comparison, 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 [Att11], [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 ϵ_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 π. 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, π 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 π 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 π by summing up to the nth 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 π at least as accurate as the one stored in the variable π?

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 nth 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.

2

Nonlinear equations

Computing the *zeros* of a real function f (equivalently, the *roots* of the equation $f(x) = 0$) is a problem that we encounter quite often in Scientific Computing. In general, this task cannot be accomplished in a finite number of operations. For instance, we have already seen in Section 1.5.1 that when f is a generic polynomial of degree greater than four, there do not exist explicit formulae for the zeros. The situation is even more difficult when f is not a polynomial.

Iterative methods are therefore adopted. Starting from one or several initial data, the methods build up a sequence of values $x^{(k)}$ that hopefully will converge to a zero α of the function f at hand.

The chapter will start with the formulation of some simple problems of practical interest, which lead to the solution of nonlinear equations. Such problems will be solved after the presentation of several numerical methods. This planning will be proposed in all the next chapters of the book.

2.1 Some representative problems

Problem 2.1 (Investment fund) At the beginning of every year a bank customer deposits v euros in an investment fund and withdraws, at the end of the nth year, a capital of M euros. We want to compute the average yearly rate of interest r of this investment. Since M is related to r by the relation

$$M = v\sum_{k=1}^{n}(1+r)^k = v\frac{1+r}{r}\left[(1+r)^n - 1\right],$$

we deduce that r is the root of the algebraic nonlinear equation:

$$f(r) = 0, \qquad \text{where } f(r) = M - v\frac{1+r}{r}[(1+r)^n - 1].$$

DOI 10.1007/978-3-642-45367-0_2, © Springer-Verlag Berlin Heidelberg 2014

This problem will be solved in Example 2.1. ■

Problem 2.2 (State equation of a gas) We want to determine the volume V occupied by a gas at temperature T and pressure p. The state equation (i.e. the equation that relates p, V and T) is

$$\left[p + a(N/V)^2\right](V - Nb) = kNT, \tag{2.1}$$

where a and b are two coefficients that depend on the specific gas, N is the number of molecules which are contained in the volume V and k is the Boltzmann constant. We need therefore to solve a nonlinear equation whose root is V (see Exercise 2.2). ■

Problem 2.3 (Rods system) Let us consider the mechanical system represented by the four rigid rods $\mathbf{a}_i$ of Figure 2.1. For any admissible value of the angle β, let us determine the value of the corresponding angle α between the rods $\mathbf{a}_1$ and $\mathbf{a}_2$. Starting from the vector identity

$$\mathbf{a}_1 - \mathbf{a}_2 - \mathbf{a}_3 - \mathbf{a}_4 = \mathbf{0}$$

and noting that the rod $\mathbf{a}_1$ is always aligned with the x-axis, we can deduce the following relationship between β and α:

$$\frac{a_1}{a_2}\cos(\beta) - \frac{a_1}{a_4}\cos(\alpha) - \cos(\beta - \alpha) = -\frac{a_1^2 + a_2^2 - a_3^2 + a_4^2}{2a_2a_4}, \tag{2.2}$$

where a_i is the known length of the ith rod. This is called the Freudenstein equation, and we can rewrite it as $f(\alpha) = 0$, where

$$f(x) = \frac{a_1}{a_2}\cos(\beta) - \frac{a_1}{a_4}\cos(x) - \cos(\beta - x) + \frac{a_1^2 + a_2^2 - a_3^2 + a_4^2}{2a_2a_4}.$$

A solution in explicit form is available only for special values of β. We would also like to mention that a solution does not exist for all values of β, and may not even be unique. To solve the equation for any given β lying between 0 and π we should invoke numerical methods (see Exercise 2.9). ■

Problem 2.4 (Population dynamics) In the study of populations (e.g. bacteria), the equation $x^+ = \phi(x) = xR(x)$ establishes a link between the number of individuals in a generation x and the number of individuals in the following generation. Function $R(x)$ models the variation rate of the considered population and can be chosen in different ways. Among the most known, we can mention:

1. Malthus's model (Thomas Malthus, 1766-1834),

$$R(x) = R_M(x) = r, \qquad r > 0;$$

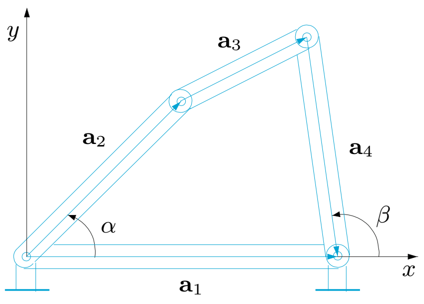


Figure 2.1. System of four rods of Problem 2.3

2. the growth with limited resources model, (known as Beverton-Holt's or discrete Verhulst's model)

$$R(x) = R_V(x) = \frac{r}{1 + xK}, \qquad r > 0, K > 0, \tag{2.3}$$

which improves on Malthus's model in considering that the growth of a population is limited by the available resources;

3. the predator/prey model with saturation,

$$R(x) = R_P = \frac{rx}{1 + (x/K)^2}, \tag{2.4}$$

which represents the evolution of Beverton-Holt's model in the presence of an antagonist population.

The dynamics of a population is therefore defined by the iterative process

$$x^{(k)} = \phi(x^{(k-1)}), \qquad k \geq 1, \tag{2.5}$$

where $x^{(k)}$ represents the number of individuals present k generations later than the initial generation $x^{(0)}$. Moreover, the stationary (or equilibrium) states x^* of the considered population are the solutions of problem

$$x^* = \phi(x^*),$$

or, equivalently, $x^* = x^* R(x^*)$ i.e. $R(x^*) = 1$. Equation (2.5) is an instance of a fixed point method (see Section 2.6). ■

2.2 The bisection method

Let f be a continuous function in $[a, b]$ which satisfies $f(a)f(b) < 0$. Then necessarily f has at least one zero in (a, b). (This result is known as the *theorem of zeros of continuous functions.*) Let us assume for simplicity

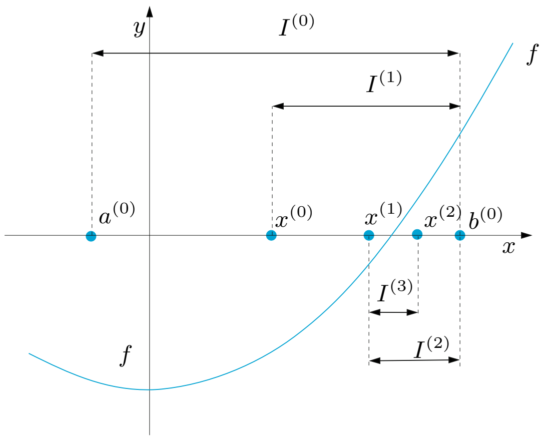


Figure 2.2. A few iterations of the bisection method

that it is unique, and let us call it α. (In the case of several zeros, by the help of the command `fplot` we can locate an interval which contains only one of them.)

The strategy of the bisection method is to halve the given interval and select that subinterval where f features a sign change. More precisely, having named $I^{(0)} = (a, b)$ and, more generally, $I^{(k)}$ the subinterval selected at step k, we choose as $I^{(k+1)}$ the sub-interval of $I^{(k)}$ at whose end-points f features a sign change. Following such procedure, it is guaranteed that every $I^{(k)}$ selected this way will contain α. The sequence $\{x^{(k)}\}$ of the midpoints of these subintervals $I^{(k)}$ will inevitably tend to α since the length of the subintervals tends to zero as k tends to infinity.
Precisely, the method is started by setting

$$a^{(0)} = a,\ b^{(0)} = b,\ I^{(0)} = (a^{(0)}, b^{(0)}),\ x^{(0)} = (a^{(0)} + b^{(0)})/2.$$

At each step $k \geq 1$ we select the subinterval $I^{(k)} = (a^{(k)}, b^{(k)})$ of the interval $I^{(k-1)} = (a^{(k-1)}, b^{(k-1)})$ as follows:

```
given $x^{(k-1)} = (a^{(k-1)} + b^{(k-1)})/2$,
if $f(x^{(k-1)}) = 0$,
   then $\alpha = x^{(k-1)}$
   and the method terminates;
otherwise,
   if $f(a^{(k-1)})f(x^{(k-1)}) < 0$
      set $a^{(k)} = a^{(k-1)},\ b^{(k)} = x^{(k-1)}$;
   if $f(x^{(k-1)})f(b^{(k-1)}) < 0$
      set $a^{(k)} = x^{(k-1)},\ b^{(k)} = b^{(k-1)}$.
```

Then we define $x^{(k)} = (a^{(k)} + b^{(k)})/2$ and increase k by 1.

For instance, in the case represented in Figure 2.2, which corresponds to the choice $f(x) = x^2 - 1$, by taking $a^{(0)} = -0.25$ and $b^{(0)} = 1.25$, we would obtain

$$\begin{array}{ll} I^{(0)} = (-0.25, 1.25), & x^{(0)} = 0.5, \\ I^{(1)} = (0.5, 1.25), & x^{(1)} = 0.875, \\ I^{(2)} = (0.875, 1.25), & x^{(2)} = 1.0625, \\ I^{(3)} = (0.875, 1.0625), & x^{(3)} = 0.96875. \end{array}$$

Notice that each subinterval $I^{(k)}$ contains the zero α. Moreover, the sequence $\{x^{(k)}\}$ necessarily converges to α since at each step the length $|I^{(k)}| = b^{(k)} - a^{(k)}$ of $I^{(k)}$ halves. Since $|I^{(k)}| = (1/2)^k |I^{(0)}|$, the error at step k satisfies

$$|e^{(k)}| = |x^{(k)} - \alpha| < \frac{1}{2}|I^{(k)}| = \left(\frac{1}{2}\right)^{k+1}(b-a).$$

In order to guarantee that $|e^{(k)}| < \varepsilon$, for a given tolerance ε it suffices to carry out k_{min} iterations, k_{min} being the smallest integer satisfying the inequality

$$\boxed{k_{min} > \log_2\left(\frac{b-a}{\varepsilon}\right) - 1} \qquad (2.6)$$

Obviously, this inequality makes sense in general, and is not confined to the specific choice of f that we have made previously.

The bisection method is implemented in Program 2.1: `fun` is the function handle associated with the function f, `a` and `b` are the endpoints of the search interval, `tol` is the tolerance ε and `nmax` is the maximum number of allowed iterations. Besides the first argument which represents the independent variable, the function `fun` can accept other auxiliary parameters.

Output parameters are `zero`, which contains the approximate value of α, the residual `res` which is the value of f in `zero` and `niter` which is the total number of iterations that are carried out. The command `find(fx==0)` finds those indices of the vector `fx` corresponding to null components, while the command `varargin` allows the function `fun` to accept a variable number of input parameters.

find
varargin

Program 2.1. bisection: bisection method

```
function [zero,res,niter]=bisection(fun,a,b,tol,...
                                    nmax,varargin)
%BISECTION Finds function zeros.
% ZERO=BISECTION(FUN,A,B,TOL,NMAX) tries to find a zero
% ZERO of the continuous function FUN in the interval
% [A,B] using the bisection method. If
% the search fails an error message is displayed.
% FUN is a function handle associated with an anonymous
% function or a Matlab function.
% ZERO=BISECTION(FUN,A,B,TOL,NMAX,P1,P2,...) passes
% parameters P1,P2,... to the function FUN(X,P1,P2,...)
% [ZERO,RES,NITER]=BISECTION(FUN,...) returns the value
% of the residual in ZERO and the iteration number at
% which ZERO was computed.
x = [a, (a+b)*0.5, b];
fx = fun(x,varargin{:});
if fx(1)*fx(3) > 0
  error([' The sign of the function at the ',...
    'endpoints of the interval must be different\n']);
  elseif fx(1) == 0
    zero = a; res = 0; niter = 0; return
elseif fx(3) == 0
    zero = b; res = 0; niter = 0; return
end
niter = 0;
I = (b - a)*0.5;
while I >= tol & niter < nmax
   niter = niter + 1;
   if fx(1)*fx(2) <  0
      x(3) = x(2);
      x(2) = x(1)+(x(3)-x(1))*0.5;
      fx = fun(x,varargin{:});
      I = (x(3)-x(1))*0.5;
   elseif fx(2)*fx(3) < 0
      x(1) = x(2);
      x(2) = x(1)+(x(3)-x(1))*0.5;
      fx = fun(x,varargin{:});
      I = (x(3)-x(1))*0.5;
   else
       x(2) = x(find(fx==0)); I = 0;
   end
end
if  (niter==nmax & I > tol)
fprintf(['Bisection stopped without converging ',...
   'to the desired tolerance because the \n',...
   'maximum number of iterations was reached\n']);
end
zero = x(2);
x = x(2);
res = fun(x,varargin{:});
return
```

Example 2.1 (Investment fund) Let us apply the bisection method to solve Problem 2.1, assuming that v is equal to 1000 euros and that after 5 years M is equal to 6000 euros. The graph of the function f can be obtained by the following instructions

```
M=6000; v=1000; f=@(r) (M-v*(1+r).*((1+r).^5 - 1)./r);
fplot(f,[0.01,0.3]);
```

(we remind the reader that the prompt is neglected in order to simplify notations). We see that f has a unique zero in the interval $(0.01, 0.1)$, which is approximately equal to 0.06. If we execute Program 2.1 with `tol`$= 10^{-12}$, `a`$= 0.01$ and `b`$= 0.1$ as follows

```
[zero,res,niter]=bisection(f,0.01,0.1,1.e-12,1000)
```

after 36 iterations the method converges to the value 0.06140241153618, in perfect agreement with the estimate (2.6) according to which $k_{min} = 36$. Thus, we conclude that the interest rate r is approximately equal to 6.14%.

Instead of an *anonymous function*, we could use the *function* `Rfuncv.m`

```
function y=Rfuncv(r,M,v)
% RFUNCV function for example 2.1
y=M - v*(1+r)./r.*((1+r).^5 - 1);
end
```

and call `bisection.m` with the following instructions:

```
M=6000; v=1000;
[zero,res,niter]=bisection(@Rfuncv,0.01,0.1,...
1.e-12,1000,M,v)
```

Notice that now the variables `M` and `v` must be appended to the input list of the function `bisection.m`. These parameters will be stored in the optional input variable `varargin` of `bisection.m`.

In spite of its simplicity, the bisection method does not guarantee a monotone reduction of the error, but simply that the search interval is halved from one iteration to the next. Consequently, if the only stopping criterion adopted is the control of the length of $I^{(k)}$, one might discard approximations of α which are quite accurate.

As a matter of fact, this method does not take into proper account the actual behavior of f. A striking fact is that it does not converge in a single iteration even if f is a linear function (unless the zero α is the midpoint of the initial search interval).

See Exercises 2.1-2.5.

2.3 The Newton method

The sign of the given function f at the endpoints of the subintervals is the only information exploited by the bisection method. A more efficient method can be constructed by exploiting the values attained by f and its derivative (in the case that f is differentiable). In that case,

$$y(x) = f(x^{(k)}) + f'(x^{(k)})(x - x^{(k)})$$

provides the equation of the tangent to the curve $(x, f(x))$ at the point $x^{(k)}$.

If we pretend that $x^{(k+1)}$ is such that $y(x^{(k+1)}) = 0$, we obtain:

$$\boxed{x^{(k+1)} = x^{(k)} - \frac{f(x^{(k)})}{f'(x^{(k)})}, \qquad k \geq 0} \tag{2.7}$$

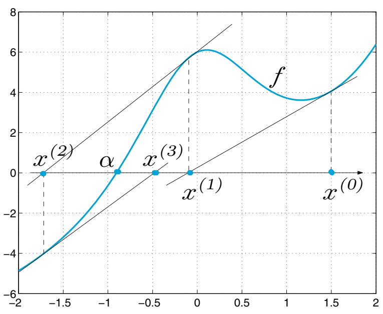


Figure 2.3. The first iterations generated by the Newton method with initial guess $x^{(0)}$ for the function $f(x) = x + e^x + 10/(1 + x^2) - 5$

provided $f'(x^{(k)}) \neq 0$. This formula allows us to compute a sequence of values $x^{(k)}$ starting from an initial guess $x^{(0)}$. This method is known as Newton's method and corresponds to computing the zero of f by locally replacing f by its tangent line (see Figure 2.3).

As a matter of fact, by developing f in Taylor series in a neighborhood of a generic point $x^{(k)}$ we find

$$f(x^{(k+1)}) = f(x^{(k)}) + \delta^{(k)} f'(x^{(k)}) + \mathcal{O}((\delta^{(k)})^2), \tag{2.8}$$

where $\delta^{(k)} = x^{(k+1)} - x^{(k)}$. Forcing $f(x^{(k+1)})$ to be zero and neglecting the term $\mathcal{O}((\delta^{(k)})^2)$, we can obtain $x^{(k+1)}$ as a function of $x^{(k)}$ as stated in (2.7). In this respect (2.7) can be regarded as an approximation of (2.8).

Obviously, (2.7) converges in a single step when f is linear, that is when $f(x) = a_1 x + a_0$.

Example 2.2 Let us solve Problem 2.1 by Newton's method, taking as initial data $x^{(0)} = 0.3$. After 6 iterations the difference between two subsequent iterates is less than or equal to 10^{-12}. ■

The Newton method in general does not converge for all possible choices of $x^{(0)}$, but only for those values of $x^{(0)}$ which are *sufficiently close* to α, that is they belong to a suitable neighbourhood $I(\alpha)$ of α. At first glance, this requirement looks meaningless: indeed, in order to compute α (which is unknown), one should start from a value sufficiently close to α!

In practice, a possible initial value $x^{(0)}$ can be obtained by resorting to a few iterations of the bisection method or, alternatively, through an investigation of the graph of f. If $x^{(0)}$ is properly chosen and α is a simple zero (that is, $f'(\alpha) \neq 0$) then the Newton method converges. Furthermore, in the special case where f is continuously differentiable up to its second derivative one has the following convergence result (see Exercise 2.8),

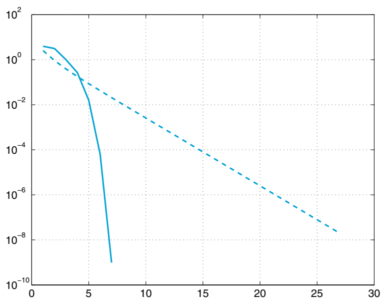


Figure 2.4. Error in semi-logarithmic scale versus iteration number for the function of Example 2.3. The dashed line corresponds to Newton's method (2.7), solid line to the modified Newton's method (2.10) (with $m = 2$)

$$\boxed{\lim_{k\to\infty} \frac{x^{(k+1)} - \alpha}{(x^{(k)} - \alpha)^2} = \frac{f''(\alpha)}{2f'(\alpha)}} \tag{2.9}$$

Consequently, if $f'(\alpha) \neq 0$ Newton's method is said to converge *quadratically*, or with order 2, since for sufficiently large values of k the error at step $(k + 1)$ behaves like the square of the error at step k multiplied by a constant which is independent of k.

In the case of zeros with multiplicity m larger than 1, i.e. if $f'(\alpha) = 0, \ldots, f^{(m-1)}(\alpha) = 0$, Newton's method still converges, but only if $x^{(0)}$ is properly chosen and $f'(x) \neq 0 \ \forall x \in I(\alpha) \setminus \{\alpha\}$. Nevertheless, in this case the order of convergence of Newton's method downgrades to 1 (see Exercise 2.15). In such case one could recover the order 2 by modifying the original method (2.7) as follows:

$$\boxed{x^{(k+1)} = x^{(k)} - m\frac{f(x^{(k)})}{f'(x^{(k)})}, \qquad k \geq 0} \tag{2.10}$$

provided that $f'(x^{(k)}) \neq 0$. Obviously, the *modified Newton's method* (2.10) requires the *a-priori* knowledge of m. If this is not the case, one could develop an *adaptive Newton method*, still of order 2, as described in [QSS07, Section 6.6.2].

Example 2.3 The function $f(x) = (x - 1)\log(x)$ has a single zero $\alpha = 1$ of multiplicity $m = 2$. Let us compute it by both Newton's method (2.7) and by its modified version (2.10). In Figure 2.4 we report the error obtained using the two methods versus the iteration number. Note that for the classical version of Newton's method the convergence is only linear. ■

2.3.1 How to terminate Newton's iterations

In theory, a convergent Newton's method returns the zero α only after an infinite number of iterations. In practice, one requires an approximation of α up to a prescribed tolerance ε. Thus the iterations can be terminated at the smallest value of k_{min} for which the following inequality holds:

$$|e^{(k_{min})}| = |\alpha - x^{(k_{min})}| < \varepsilon.$$

This is a test on the error. Unfortunately, since the error is unknown, one needs to adopt in its place a suitable *error estimator*, that is, a quantity that can be easily computed and through which we can estimate the real error. At the end of Section 2.6, we will see that a suitable error estimator for Newton's method is provided by the difference between two successive iterates. This means that one terminates the iterations at step k_{min} as soon as

$$\boxed{|x^{(k_{min})} - x^{(k_{min}-1)}| < \varepsilon} \qquad (2.11)$$

This is a test on the increment. We will see in Section 2.6.1 that the test on the increment is satisfactory when α is a simple zero of f. Alternatively, one could use a test on the *residual* at step k, $r^{(k)} = f(x^{(k)})$ (note that the residual is null when $x^{(k)}$ is a zero of the function f).

Precisely, we could stop the iteration at the first k_{min} for which

$$\boxed{|r^{(k_{min})}| = |f(x^{(k_{min})})| < \varepsilon} \qquad (2.12)$$

The test on the residual is satisfactory only when $|f'(x)| \simeq 1$ in a neighborhood I_α of the zero α (see Figure 2.5). Otherwise, it will produce an over estimation of the error if $|f'(x)| \gg 1$ for $x \in I_\alpha$ and an under estimation if $|f'(x)| \ll 1$ (see also Exercise 2.6).

In Program 2.2 we implement Newton's method (2.7). Its modified form can be obtained simply by replacing f' with f'/m. The input parameters `fun` and `dfun` are the function handles associated with the function f and its first derivative, while `x0` is the initial guess. The method will be terminated when the absolute value of the difference between two subsequent iterates is less than the prescribed tolerance `tol`, or when the maximum number of iterations `nmax` has been reached.

Program 2.2. newton: Newton method

```
function [zero,res,niter]=newton(fun,dfun,x0,tol,...
                                 nmax,varargin)
%NEWTON Finds function zeros.
% ZERO=NEWTON(FUN,DFUN,X0,TOL,NMAX) tries to find the
% zero ZERO of the continuous and differentiable
```

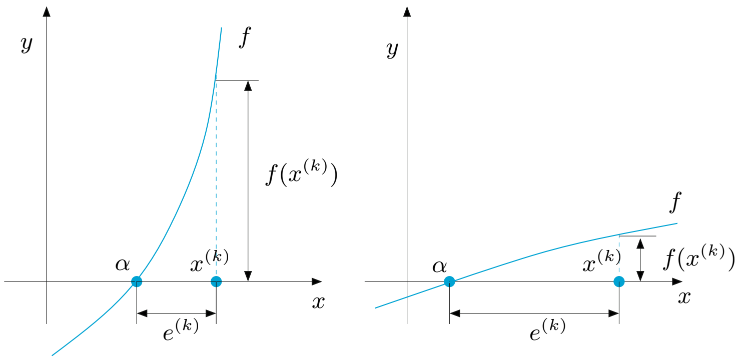


Figure 2.5. Two situations in which the residual is a poor error estimator: $|f'(x)| \gg 1$ (*left*), $|f'(x)| \ll 1$ (*right*), with x belonging to a neighborhood of α

```
% function FUN nearest to X0 using the Newton method.
% FUN and its derivative DFUN accept real scalar input
% x and return a real scalar value. If the search
% fails an error message is displayed. FUN and DFUN
% are function handles associated with anonymous fun-
% ctions or Matlab functions.
% ZERO=NEWTON(FUN,DFUN,X0,TOL,NMAX,P1,P2,...) passes
% parameters P1,P2,... to functions: FUN(X,P1,P2,...)
% and DFUN(X,P1,P2,...).
% [ZERO,RES,NITER]=NEWTON(FUN,...) returns the value of
% the residual in ZERO and the iteration number at
% which ZERO was computed.
x = x0; fx = fun(x,varargin{:});
dfx = dfun(x,varargin{:});
niter = 0; diff = tol+1;
while diff >= tol & niter < nmax
   niter = niter + 1;       diff = - fx/dfx;
   x = x + diff;            diff = abs(diff);
   fx = fun(x,varargin{:});
   dfx = dfun(x,varargin{:});
end
if (niter==nmax & diff > tol)
    fprintf(['Newton stopped without converging to',...
    ' the desired tolerance because the maximum\n ',...
    'number of iterations was reached\n']);
end
zero = x; res = fx;
```

2.4 The secant method

For the computation of the zeroes of a function f whose derivative is not available in analytical form, the Newton method cannot be applied. However, should we be able to compute the function f at any arbitrary point, we could replace the exact value $f'(x^{(k)})$ with an incremental ratio based on previously computed values of f. The secant method exploits this strategy and is defined as follows: for any given couple of points $x^{(0)}$ and $x^{(1)}$, for $k \geq 1$ compute

$$\boxed{x^{(k+1)} = x^{(k)} - \left(\frac{f(x^{(k)}) - f(x^{(k-1)})}{x^{(k)} - x^{(k-1)}} \right)^{-1} f(x^{(k)})} \qquad (2.13)$$

If α is a simple zero of f and $I(\alpha)$ a suitable neighborhood of α, if moreover $x^{(0)}$ and $x^{(1)}$ are sufficiently close to α and $f'(x) \neq 0$ $\forall x \in I(\alpha) \setminus \{\alpha\}$, the secant method (2.13) converges to α. Moreover, if $f \in C^2(I(\alpha))$ and $f'(\alpha) \neq 0$, there exists a constant $c > 0$ such that

$$\boxed{|x^{(k+1)} - \alpha| \leq c|x^{(k)} - \alpha|^p, \text{ with } p = \frac{1+\sqrt{5}}{2} \simeq 1.618...} \qquad (2.14)$$

This shows that the secant method converges *super-linearly* (i.e. with a convergence rate $p > 1$). Should α be a multiple zero, the rate of convergence would be linear, just as for Newton's method.

Example 2.4 We use the secant method to solve the case of Example 2.1 starting from the initial data $x^{(0)} = 0.3$ and $x^{(1)} = -0.3$. The method converges in 8 iterations, whereas 6 iterations would be necessary to the Newton method to converge starting from the same $x^{(0)}$.
Choosing instead $x^{(0)} = 0.3$ and $x^{(1)} = 0.1$ the secant method would converge in 6 iterations, just like the Newton method. ■

2.5 Systems of nonlinear equations

Let us consider a system of nonlinear equations of the form

$$\begin{cases} f_1(x_1, x_2, \ldots, x_n) = 0, \\ f_2(x_1, x_2, \ldots, x_n) = 0, \\ \vdots \\ f_n(x_1, x_2, \ldots, x_n) = 0, \end{cases} \qquad (2.15)$$

where $f_1, \ldots, f_n$ are nonlinear functions. Setting $\mathbf{f} = (f_1, \ldots, f_n)^T$ and $\mathbf{x} = (x_1, \ldots, x_n)^T$, system (2.15) can be written in a compact way as

$$\mathbf{f}(\mathbf{x}) = \mathbf{0}. \qquad (2.16)$$

An example is given by the following nonlinear system

$$\begin{cases} f_1(x_1,x_2) = x_1^2 + x_2^2 = 1, \\ f_2(x_1,x_2) = \sin(\pi x_1/2) + x_2^3 = 0. \end{cases} \qquad (2.17)$$

In order to extend Newton's method to the case of a system, we replace the first derivative of the scalar function f with the *Jacobian matrix* $\mathrm{J}_\mathbf{f}$ of the vectorial function $\mathbf{f}$ whose components are

$$(\mathrm{J}_\mathbf{f})_{ij} = \frac{\partial f_i}{\partial x_j}, \qquad i,j = 1,\ldots,n.$$

The symbol $\partial f_i/\partial x_j$ represents the partial derivative of f_i with respect to x_j (see definition (9.3)). With this notation, Newton's method for (2.16) then becomes: given $\mathbf{x}^{(0)} \in \mathbb{R}^n$, for $k = 0, 1, \ldots$, until convergence

$$\boxed{\begin{array}{ll} \text{solve} & \mathrm{J}_\mathbf{f}(\mathbf{x}^{(k)})\boldsymbol{\delta}\mathbf{x}^{(k)} = -\mathbf{f}(\mathbf{x}^{(k)}) \\ \text{set} & \mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \boldsymbol{\delta}\mathbf{x}^{(k)} \end{array}} \qquad (2.18)$$

Therefore, Newton's method applied to a system requires at each step the solution of a linear system with matrix $\mathrm{J}_\mathbf{f}(\mathbf{x}^{(k)})$.

Program 2.3 implements this method by using the MATLAB command \ (see Section 5.8) to solve the linear system with the jacobian matrix. In input we must define a column vector `x0` representing the initial datum and two *functions*, the function handles `Ffun` and `Jfun`, to evaluate (respectively) the function $\mathbf{f}$ and the jacobian matrix $\mathrm{J}_\mathbf{f}$, for a generic vector `x`. The method stops when the difference between two consecutive iterates has an euclidean norm smaller than `tol` or when `nmax`, the maximum number of allowed iterations, has been reached.

Program 2.3. newtonsys: Newton method for nonlinear systems

```
function [x,res,niter] = newtonsys(Ffun,Jfun,x0,tol,...
                                   nmax, varargin)
%NEWTONSYS Finds a zero of a nonlinear system
% [ZERO,RES,NITER]=NEWTONSYS(FFUN,JFUN,X0,TOL,NMAX)
% tries to find the vector ZERO, zero of a nonlinear
% system defined in FFUN with jacobian matrix defined
% in the function JFUN, nearest to the vector X0.
% The variable RES returns the residual in ZERO
% while NITER returns the number of iterations needed
% to compute ZERO. FFUN and JFUN are function handles
% associated with anonymous functions or MATLAB
% functions stored in M-files.
niter = 0; err = tol + 1; x = x0;
while err >= tol & niter < nmax
    J = Jfun(x,varargin{:});
    F = Ffun(x,varargin{:});
    delta = - J\F;
    x = x + delta;
    err = norm(delta);
    niter = niter + 1;
end
res = norm(Ffun(x,varargin{:}));
if (niter==nmax & err> tol)
 fprintf([' Fails to converge within maximum ',...
         'number of iterations.\n',...
         'The iterate returned has relative ',...
         'residual %e\n'],F);
else
 fprintf(['The method converged at iteration ',...
         '%i with  residual %e\n'],niter,F);
end
return
```

Example 2.5 Let us consider the nonlinear system (2.17) which allows the two (graphically detectable) solutions $(0.4761, -0.8794)$ and $(-0.4761, 0.8794)$ (where we only report the four first significant digits). In order to use Program 2.3 we define the following *functions*

```
function J=Jfun(x)
pi2 = 0.5*pi;
J(1,1) = 2*x(1);
J(1,2) = 2*x(2);
J(2,1) = pi2*cos(pi2*x(1));
J(2,2) = 3*x(2)^2;
return

function F=Ffun(x)
F(1,1) = x(1)^2 + x(2)^2 - 1;
F(2,1) = sin(pi*x(1)/2) + x(2)^3;
return
```

Starting from an initial datum of `x0=[1;1]` Newton's method, launched with the command

```
x0=[1;1]; tol=1e-5; nmax=10;
[x,F,niter] = newtonsys(@Ffun,@Jfun,x0,tol,nmax);
```

converges in 8 iterations to the values

```
4.760958225338114e-01
-8.793934089897496e-01
```

(The special character `@` generates the function handles associated with functions `Ffun` and `Jfun`, which are passed to `newtonsys`.)

Notice that the method converges to the other root starting from `x0=[-1;-1]`. In general, exactly as in the case of scalar functions, convergence of Newton's method will actually depend on the choice of the initial datum $\mathbf{x}^{(0)}$ and in particular we should guarantee that $\det(\mathrm{J}_{\mathbf{f}}(\mathbf{x}^{(0)})) \neq 0$. ■

The secant method can be adapted to the solution of systems of nonlinear equations still featuring super-linear rate of convergence. The idea consists in replacing the Jacobian matrices $\mathrm{J}_{\mathbf{f}}(\mathbf{x}^{(k)})$ (for $k \geq 0$) of Newton's method with suitable matrices B_k, recursively defined starting from a convenient matrix B_0, representing a suitable approximation of $\mathrm{J}_{\mathbf{f}}(\mathbf{x}^{(0)})$. (Alternative strategies will be addressed in Section 4.2 and in Chapter 9.) The most popular method of this kind is the following Broyden method. From a given $\mathbf{x}^{(0)} \in \mathbb{R}^n$ and a given $\mathrm{B}_0 \in \mathbb{R}^{n\times n}$, for $k = 0, 1, \ldots$, until convergence

$$\boxed{\begin{array}{ll} \text{solve} & \mathrm{B}_k \boldsymbol{\delta}\mathbf{x}^{(k)} = -\mathbf{f}(\mathbf{x}^{(k)}) \\ \text{set} & \mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \boldsymbol{\delta}\mathbf{x}^{(k)} \\ \text{set} & \boldsymbol{\delta}\mathbf{f}^{(k)} = \mathbf{f}(\mathbf{x}^{(k+1)}) - \mathbf{f}(\mathbf{x}^{(k)}) \\ \text{compute} & \mathrm{B}_{k+1} = \mathrm{B}_k + \dfrac{(\boldsymbol{\delta}\mathbf{f}^{(k)} - \mathrm{B}_k\boldsymbol{\delta}\mathbf{x}^{(k)})\boldsymbol{\delta}\mathbf{x}^{(k)T}}{\boldsymbol{\delta}\mathbf{x}^{(k)T}\boldsymbol{\delta}\mathbf{x}^{(k)}} \end{array}} \tag{2.19}$$

Notice that we do not require the sequence $\{\mathrm{B}_k\}$ to converge to the true Jacobian matrix $\mathrm{J}_{\mathbf{f}}(\boldsymbol{\alpha})$ ($\boldsymbol{\alpha}$ being the root of the system). Rather, it can be proved that

$$\lim_{k\to\infty} \frac{\|(\mathrm{B}_k - \mathrm{J}_{\mathbf{f}}(\boldsymbol{\alpha}))(\mathbf{x}^{(k)} - \boldsymbol{\alpha})\|}{\|\mathbf{x}^{(k)} - \boldsymbol{\alpha}\|} = 0.$$

This property guarantees that B_k is a convenient approximation of $\mathrm{J}_{\mathbf{f}}(\boldsymbol{\alpha})$ along the error direction $\mathbf{x}^{(k)} - \boldsymbol{\alpha}$.

At every step, the virtual cost $\mathcal{O}(n^3)$ for the computation of $\boldsymbol{\delta}\mathbf{x}^{(k)}$ can be reduced to $\mathcal{O}(n^2)$, by recursively using QR factorization of matrices B_k (see e.g., [GM72]). Because of the equality $(\boldsymbol{\delta}\mathbf{f}^{(k)} - \mathrm{B}_k\boldsymbol{\delta}\mathbf{x}^{(k)}) = \mathbf{f}(\mathbf{x}^{(k+1)})$, we can avoid implementing matrix-vector products for the computation of B_{k+1}.

For a more in depth description of Broyden method and other secant type methods, also called *quasi-Newton* methods, we refer to [JS96] [Deu04], [SM03] and [QSS07, Ch. 6].

Example 2.6 Let us use Broyden method (2.19) for the solution of the problem of Example 2.5. Setting $\mathrm{B}_0 = \mathrm{I}$, the tolerance $\varepsilon = 10^{-5}$ for the stopping test on the increment and $\mathbf{x}^{(0)} = (1, 1)^T$, convergence to the point $(0.476095825652119, -0.879393405072448)^T$ is achieved in 10 iterations, with a residual whose norm is $1.324932e-08$. The Newton method would converge in 8 iterations with the norm of the final residual equal to $2.235421e-11$. Still using $\mathrm{B}_0 = \mathrm{I}$ and changing the initial point $\mathbf{x}^{(0)} = (-1, -1)^T$, convergence would be achieved in 17 iterations with a (norm of the) residual equal to $5.744382e-08$ (versus 8 iterations and residual $2.235421e-11$ for Newton's method). Choosing instead $B_0 = 2I$, the number of Broyden iterations reduces to 12; this shows how crucial is the choice of the initial matrix.

As for accuracy, Newton's method is better than Broyden's, as the value of the residual shows. ■

Let us summarize

1. Methods for the computation of the zeros of a function f are usually of iterative type;
2. the bisection method computes a zero of a function f by generating a sequence of intervals whose length is halved at each iteration. This method is convergent provided that f is continuous in the initial interval and has opposite signs at the endpoints of this interval;
3. Newton's method computes a zero α of f by taking into account the values of f and of its derivative. A necessary condition for convergence is that the initial datum belongs to a suitable (sufficiently small) neighborhood of α;
4. Newton's method is quadratically convergent only when α is a simple zero of f, otherwise convergence is linear;
5. the Newton method can be extended to the case of a nonlinear system of equations;
6. the secant method can be regarded as a variant of Newton's method where the first derivative of the function is replaced by a suitable incremental ratio. For simple roots, it converges super-linearly (however less than quadratically); for multiple roots, the convergence rate is only linear. As for Newton's method, the initial points should be sufficiently close to the root for the method to converge.

See Exercises 2.6-2.14.

2.6 Fixed point iterations

Playing with a pocket calculator, one may verify that by applying repeatedly the cosine key to the real value 1, one gets the following sequence of real numbers:

$$
\begin{aligned}
&x^{(1)} = \cos(1) = 0.54030230586814,\\
&x^{(2)} = \cos(x^{(1)}) = 0.85755321584639,\\
&\vdots\\
&x^{(10)} = \cos(x^{(9)}) = 0.74423735490056,\\
&\vdots\\
&x^{(20)} = \cos(x^{(19)}) = 0.73918439977149,
\end{aligned}
$$

which should tend to the value $\alpha = 0.73908513\ldots$. Since, by construction, $x^{(k+1)} = \cos(x^{(k)})$ for $k = 0, 1, \ldots$ (with $x^{(0)} = 1$), the limit α satisfies the equation $\cos(\alpha) = \alpha$. For this reason α is called a fixed

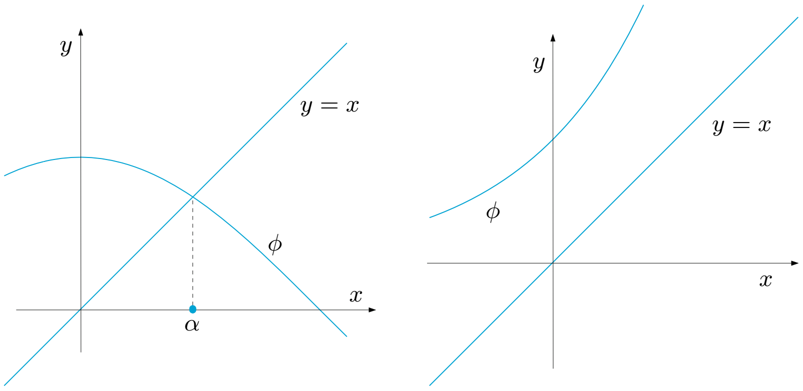


Figure 2.6. The function $\phi(x) = \cos x$ admits one and only one fixed point (*left*), whereas the function $\phi(x) = e^x$ does not have any (*right*)

point of the cosine function. We may wonder how such iterations could be exploited in order to compute the zeros of a given function. In the previous example, α is not only a fixed point for the cosine function, but also a zero of the function $f(x) = x - \cos(x)$, hence the previously proposed method can be regarded as a method to compute the zeros of f. On the other hand, not every function has fixed points. For instance, by repeating the previous experiment using the exponential function and $x^{(0)} = 1$ one encounters a situation of overflow after 4 steps only (see Figure 2.6).

Let us clarify the intuitive idea above by considering the following problem. Given a function $\phi : [a, b] \to \mathbb{R}$, find $\alpha \in [a, b]$ such that

$$\alpha = \phi(\alpha).$$

If such an α exists it will be called a *fixed point* of ϕ and it could be computed by the following algorithm:

$$\boxed{x^{(k+1)} = \phi(x^{(k)}), \qquad k \geq 0} \tag{2.20}$$

where $x^{(0)}$ is an initial guess. This algorithm is called *fixed point iterations* and ϕ is said to be the *iteration function*. The introductory example is therefore an instance of fixed point iterations with $\phi(x) = \cos(x)$.

A geometrical interpretation of (2.20) is provided in Figure 2.7 (*left*). One can guess that if ϕ is a continuous function and the limit of the sequence $\{x^{(k)}\}$ exists, then such limit is a fixed point of ϕ. We will make this result more precise in Propositions 2.1 and 2.2.

Example 2.7 The Newton method (2.7) can be regarded as an algorithm of fixed point iterations whose iteration function is

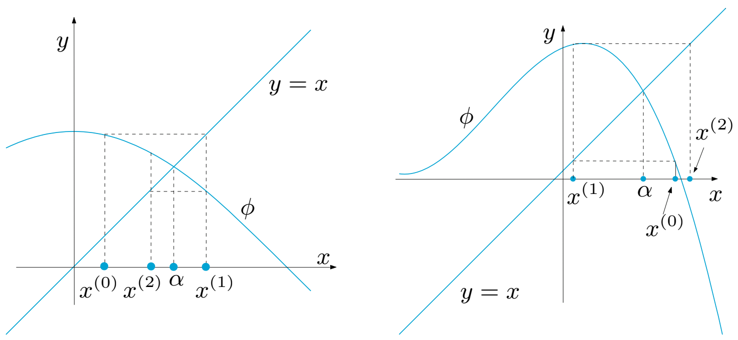


Figure 2.7. Representation of a few fixed point iterations for two different iteration functions. At left, the iterations converge to the fixed point α, whereas the iterations on the right produce a divergence sequence

$$\phi(x) = x - \frac{f(x)}{f'(x)}. \tag{2.21}$$

From now on this function will be denoted by ϕ_N (where N stands for Newton). This is not the case for the bisection method since the generic iterate $x^{(k+1)}$ depends not only on $x^{(k)}$ but also on $x^{(k-1)}$. ■

As shown in Figure 2.7 (*right*), fixed point iterations may not converge. Indeed, the following result holds.

Proposition 2.1 *Let us consider the sequence* (2.20)*.*

1. *Let us suppose that $\phi(x)$ is continuous in $[a, b]$ and such that $\phi(x) \in [a, b]$ for every $x \in [a, b]$; then there exists at least a fixed point $\alpha \in [a, b]$.*
2. *Moreover, if*

$$\exists L < 1 \ s.t. \ |\phi(x_1) - \phi(x_2)| \leq L|x_1 - x_2| \ \ \forall x_1, x_2 \in [a, b], \tag{2.22}$$

then there exists a unique fixed point $\alpha \in [a, b]$ of ϕ and the sequence defined in (2.20) *converges to α, for any choice of intial guess $x^{(0)}$ in $[a, b]$.*

Proof. *1.* We start by proving existence of fixed points for ϕ. The function $g(x) = \phi(x) - x$ is continuous in $[a, b]$ and, thanks to assumption made on the range of ϕ, it holds $g(a) = \phi(a) - a \geq 0$ and $g(b) = \phi(b) - b \leq 0$. By applying the theorem of zeros of continuous functions, we can conclude that g has at least one zero in $[a, b]$, i.e. ϕ has at least one fixed point in $[a, b]$. (See Figure 2.8 for an instance.)

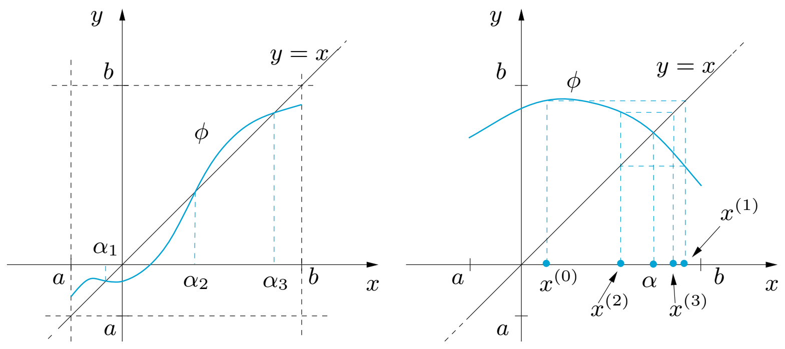


Figure 2.8. At left, an iteration function ϕ featuring 3 fixed points, at right, an iteration function satisfying the assumption (2.22) and the first elements of sequence (2.24) converging to the unique fixed point α

2. Uniqueness of fixed points follows from assumption (2.22). Indeed, should two different fixed points α_1 and α_2 exist, then

$$|\alpha_1 - \alpha_2| = |\phi(\alpha_1) - \phi(\alpha_2)| \leq L|\alpha_1 - \alpha_2| < |\alpha_1 - \alpha_2|,$$

which cannot be.
We prove now that the sequence $x^{(k)}$ defined in (2.20) converges to the unique fixed point α when $k \to \infty$, for any choice of initial guess $x^{(0)} \in [a, b]$. It holds

$$\begin{aligned} 0 \leq |x^{(k+1)} - \alpha| &= |\phi(x^{(k)}) - \phi(\alpha)| \\ &\leq L|x^{(k)} - \alpha| \leq \ldots \leq L^{k+1}|x^{(0)} - \alpha|, \end{aligned}$$

i.e., $\forall k \geq 0$,

$$\frac{|x^{(k)} - \alpha|}{|x^{(0)} - \alpha|} \leq L^k. \tag{2.23}$$

Passing to the limit as $k \to \infty$, we obtain $\lim_{k\to\infty} |x^{(k)} - \alpha| = 0$, which is the desired result. ∎

In practice it is often very difficult to choose *a priori* an interval $[a, b]$ for which the assumptions of Proposition 2.1 are fulfilled; in such cases the following *local* convergence result will be useful. We refer to [OR70] for a proof.

Theorem 2.1 (Ostrowski's theorem) *Let α be a fixed point of a function ϕ which is continuous and continuously differentiable in a suitable neighbourhood $\mathcal{J}$ of α. If $|\phi'(\alpha)| < 1$, then there exists $\delta > 0$ for which $\{x^{(k)}\}$ converges to α, for every $x^{(0)}$ such that $|x^{(0)} - \alpha| < \delta$. Moreover, it holds*

$$\boxed{\lim_{k\to\infty} \frac{x^{(k+1)} - \alpha}{x^{(k)} - \alpha} = \phi'(\alpha)} \qquad (2.24)$$

Proof. We limit ourselves to verify property (2.24). Thanks to Lagrange theorem, for any $k \geq 0$, there exists a point ξ_k between $x^{(k)}$ and α such that $x^{(k+1)} - \alpha = \phi(x^{(k)}) - \phi(\alpha) = \phi'(\xi_k)(x^{(k)} - \alpha)$, that is

$$(x^{(k+1)} - \alpha)/(x^{(k)} - \alpha) = \phi'(\xi_k). \qquad (2.25)$$

Since $x^{(k)} \to \alpha$ and ξ_k lies between $x^{(k)}$ and α, it holds $\lim_{k\to\infty} \xi_k = \alpha$. Finally, passing to the limit in both terms of (2.25) and recalling that ϕ' is continuous in a neighbourhood of α, we obtain (2.24). ■

From both (2.23) and (2.24) one deduces that the fixed point iterations converge at least linearly, that is, for k sufficiently large the error at step $k+1$ behaves like the error at step k multiplied by a constant (which concides with either L in (2.23) and $\phi'(\alpha)$ in (2.24)) which is independent of k and whose absolute value is strictly less than 1. For this reason, this constant is named *asymptotic convergence factor*. Finally, we remark that the smaller the *asymptotic convergence factor*, the faster the convergence.

Remark 2.1 When $|\phi'(\alpha)| > 1$, it follows from (2.25) that if $x^{(k)}$ is sufficiently close to α, such that $|\phi'(x^{(k)})| > 1$, then $|\alpha - x^{(k+1)}| > |\alpha - x^{(k)}|$, and the sequence cannot converge to the fixed point. On the contrary, when $|\phi'(\alpha)| = 1$, no conclusion can be drawn since either convergence or divergence could take place, depending on properties of the iteration function $\phi(x)$. ■

Example 2.8 The function $\phi(x) = \cos(x)$ satisfies all the assumptions of Theorem 2.1. Indeed, $|\phi'(\alpha)| = |\sin(\alpha)| \simeq 0.67 < 1$, and thus by continuity there exists a neighborhood I_α of α such that $|\phi'(x)| < 1$ for all $x \in I_\alpha$. The function $\phi(x) = x^2 - 1$ has two fixed points $\alpha_\pm = (1 \pm \sqrt{5})/2$, however it does not satisfy the assumption for either since $|\phi'(\alpha_\pm)| = |1 \pm \sqrt{5}| > 1$. The corresponding fixed point iterations will not converge. ■

Example 2.9 (Population dynamics) Let us apply the fixed point iterations to the function $\phi_V(x) = rx/(1 + xK)$ of Verhulst's model (2.3) and to the function $\phi_P(x) = rx^2/(1 + (x/K)^2)$, for $r = 3$ and $K = 1$, of the predator/prey model (2.4). Starting from the initial point $x^{(0)} = 1$, we find the fixed

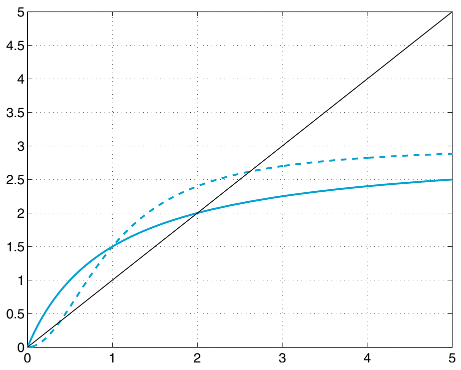


Figure 2.9. Two fixed points for two different population dynamics: Verhulst's model (*solid line*) and predator/prey model (*dashed line*)

point $\alpha = 2$ in the first case and $\alpha = 2.6180$ in the second case (see Figure 2.9). The fixed point $\alpha = 0$, common to either ϕ_V and ϕ_P, can be obtained using the fixed point iterations on ϕ_P but not those on ϕ_V. In fact, $\phi_P'(\alpha) = 0$, while $\phi_V'(\alpha) = r > 1$. The third fixed point of ϕ_P, $\alpha = 0.3820\ldots$, cannot be obtained by fixed point iterations since $\phi_P'(\alpha) > 1$. ■

The Newton method is not the only iterative procedure featuring quadratic convergence. Indeed, the following general property holds.

Proposition 2.2 *Assume that all hypotheses of Theorem 2.1 are satisfied. In addition assume that ϕ is twice continuously differentiable and that*

$$\phi'(\alpha) = 0,\ \phi''(\alpha) \neq 0.$$

Then the fixed point iterations (2.20) converge with order 2 and

$$\boxed{\lim_{k\to\infty} \frac{x^{(k+1)} - \alpha}{(x^{(k)} - \alpha)^2} = \frac{1}{2}\phi''(\alpha)} \qquad (2.26)$$

Proof. In this case it suffices to prove that there exists a point $\eta^{(k)}$ lying between $x^{(k)}$ and α such that

$$x^{(k+1)} - \alpha = \phi(x^{(k)}) - \phi(\alpha) = \phi'(\alpha)(x^{(k)} - \alpha) + \frac{\phi''(\eta^{(k)})}{2}(x^{(k)} - \alpha)^2.$$

■

Example 2.7 shows that the fixed point iterations (2.20) could also be used to compute the zeros of the function f. Clearly for any given f the function ϕ defined in (2.21) is not the only possible iteration function. For instance, for the solution of the equation $\log(x) = \gamma$, after setting $f(x) = \log(x) - \gamma$, the choice (2.21) could lead to the iteration function

$$\phi_N(x) = x(1 - \log(x) + \gamma).$$

Another fixed point iteration algorithm could be obtained by adding x to both sides of the equation $f(x) = 0$. The associated iteration function is now $\phi_1(x) = x + \log(x) - \gamma$. A further method could be obtained by choosing the iteration function $\phi_2(x) = x\log(x)/\gamma$. Not all these methods are convergent. For instance, if $\gamma = -2$, the methods corresponding to the iteration functions ϕ_N and ϕ_2 are both convergent, whereas the one corresponding to ϕ_1 is not since $|\phi_1'(x)| > 1$ in a neighborhood of the fixed point α.

2.6.1 How to terminate fixed point iterations

In general, fixed point iterations are terminated when the absolute value of the difference between two consecutive iterates is less than a prescribed tolerance ε.

Since $\alpha = \phi(\alpha)$ and $x^{(k+1)} = \phi(x^{(k)})$, using the mean value theorem (see Section 1.5.3) we find

$$\alpha - x^{(k+1)} = \phi(\alpha) - \phi(x^{(k)}) = \phi'(\xi^{(k)})\ (\alpha - x^{(k)}) \quad \text{with } \xi^{(k)} \in I_{\alpha, x^{(k)}},$$

$I_{\alpha, x^{(k)}}$ being the interval with endpoints α and $x^{(k)}$. Using the identity

$$\alpha - x^{(k)} = (\alpha - x^{(k+1)}) + (x^{(k+1)} - x^{(k)}),$$

it follows that

$$\alpha - x^{(k)} = \frac{1}{1 - \phi'(\xi^{(k)})}(x^{(k+1)} - x^{(k)}). \tag{2.27}$$

Consequently, if $\phi'(x) \simeq 0$ in a neighborhood of α, the difference between two consecutive iterates provides a satisfactory error estimator. This is the case for methods of order 2, including Newton's method. This estimate becomes the more unsatisfactory the more ϕ' approaches 1.

Example 2.10 Let us compute with Newton's method the zero $\alpha = 1$ of the function $f(x) = (x-1)^{m-1}\log(x)$ for $m = 11$ and $m = 21$, whose multiplicity is equal to m. In this case Newton's method converges with order 1; moreover, it is possible to prove (see Exercise 2.15) that $\phi_N'(\alpha) = 1 - 1/m$, ϕ_N being the iteration function of the method, regarded as a fixed point iteration algorithm. As m increases, the accuracy of the error estimate provided by the difference between two consecutive iterates decreases. This is confirmed by the numerical results in Figure 2.10 where we compare the behavior of the true error with that of our estimator for both $m = 11$ and $m = 21$. The difference between these two quantities is greater for $m = 21$. ■

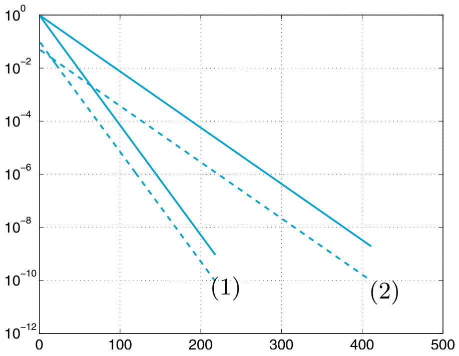


Figure 2.10. Absolute values of the errors (*solid line*) and absolute values of the difference between two consecutive iterates (*dashed line*), plotted versus the number of iterations for the case of Example 2.10. Graphs (1) refer to $m = 11$, graphs (2) to $m = 21$

2.7 Acceleration using Aitken method

In this paragraph we will illustrate a technique which allows to accelerate the convergence of a sequence obtained via fixed point iterations. Therefore, we suppose that $x^{(k)} = \phi(x^{(k-1)})$, $k \geq 1$. If the sequence $\{x^{(k)}\}$ converges *linearly* to a fixed point α of ϕ, we have from (2.24) that, for a given k, there must be a value λ (to be determined) such that

$$\phi(x^{(k)}) - \alpha = \lambda(x^{(k)} - \alpha), \tag{2.28}$$

where we have deliberately avoided to identify $\phi(x^{(k)})$ with $x^{(k+1)}$. Indeed, the idea underlying Aitken's method consists in defining a new value for $x^{(k+1)}$ (and thus a new sequence) which is a better approximation for α than that given by $\phi(x^{(k)})$. As a matter of fact, from (2.28) we have that

$$\alpha = \frac{\phi(x^{(k)}) - \lambda x^{(k)}}{1 - \lambda} = \frac{\phi(x^{(k)}) - \lambda x^{(k)} + x^{(k)} - x^{(k)}}{1 - \lambda}$$

or

$$\boxed{\alpha = x^{(k)} + (\phi(x^{(k)}) - x^{(k)})/(1 - \lambda)} \tag{2.29}$$

We must now compute λ. To do so, we introduce the following sequence

$$\lambda^{(k)} = \frac{\phi(\phi(x^{(k)})) - \phi(x^{(k)})}{\phi(x^{(k)}) - x^{(k)}} \tag{2.30}$$

and verify that the following property holds:

Lemma 2.1 *If the sequence of elements $x^{(k+1)} = \phi(x^{(k)})$ converges to α, then $\lim\limits_{k\to\infty} \lambda^{(k)} = \phi'(\alpha)$.*

Proof. If $x^{(k+1)} = \phi(x^{(k)})$, then $x^{(k+2)} = \phi(\phi(x^{(k)}))$ and from (2.30), we obtain that $\lambda^{(k)} = (x^{(k+2)} - x^{(k+1)})/(x^{(k+1)} - x^{(k)})$ or

$$\lambda^{(k)} = \frac{x^{(k+2)} - \alpha - (x^{(k+1)} - \alpha)}{x^{(k+1)} - \alpha - (x^{(k)} - \alpha)} = \frac{\dfrac{x^{(k+2)} - \alpha}{x^{(k+1)} - \alpha} - 1}{1 - \dfrac{x^{(k)} - \alpha}{x^{(k+1)} - \alpha}}$$

from which, computing the limit and recalling (2.24), we find

$$\lim_{k\to\infty} \lambda^{(k)} = \frac{\phi'(\alpha) - 1}{1 - 1/\phi'(\alpha)} = \phi'(\alpha).$$

■

Thanks to Lemma 2.1 we can conclude that, for a given k, $\lambda^{(k)}$ can be considered as an approximation of the previously introduced unknown value λ. Thus, we use (2.30) in (2.29) and define a new $x^{(k+1)}$ as follows:

$$\boxed{x^{(k+1)} = x^{(k)} - \frac{(\phi(x^{(k)}) - x^{(k)})^2}{\phi(\phi(x^{(k)})) - 2\phi(x^{(k)}) + x^{(k)}}, \ k \geq 0} \tag{2.31}$$

This expression is known as *Aitken's extrapolation formula* and it can be considered as a *new* fixed point iteration for the new iteration function

$$\phi_\Delta(x) = \frac{x\phi(\phi(x)) - [\phi(x)]^2}{\phi(\phi(x)) - 2\phi(x) + x}.$$

This method is sometimes called *Steffensen's method*. Clearly, function ϕ_Δ is undetermined for $x = \alpha$ as the numerator and denominator vanish. However, by applying de l'Hôpital's formula and assuming that ϕ is differentiable with $\phi'(\alpha) \neq 1$ one finds

$$\begin{aligned}\lim_{x\to\alpha} \phi_\Delta(x) &= \frac{\phi(\phi(\alpha)) + \alpha\phi'(\phi(\alpha))\phi'(\alpha) - 2\phi(\alpha)\phi'(\alpha)}{\phi'(\phi(\alpha))\phi'(\alpha) - 2\phi'(\alpha) + 1} \\ &= \frac{\alpha + \alpha[\phi'(\alpha)]^2 - 2\alpha\phi'(\alpha)}{[\phi'(\alpha)]^2 - 2\phi'(\alpha) + 1} = \alpha.\end{aligned}$$

Consequently, $\phi_\Delta(x)$ can be extended by continuity to $x = \alpha$ by setting $\phi_\Delta(\alpha) = \alpha$.

When $\phi(x) = x - f(x)$, the case $\phi'(\alpha) = 1$ corresponds to a root with multiplicity of at least 2 for f (since $\phi'(\alpha) = 1 - f'(\alpha)$). In such situation however, we can once again prove by evaluating the limit that $\phi_\Delta(\alpha) = \alpha$. Moreover, we can also verify that the fixed points of ϕ_Δ are all and exclusively the fixed points of ϕ.

Aitken's method can thus be applied for any fixed point method. Indeed, the following theorem holds:

Theorem 2.2 *Let $x^{(k+1)} = \phi(x^{(k)})$ be the fixed point iterations (2.20) with $\phi(x) = x - f(x)$ for computing the roots of f. Then if f is sufficiently regular we have:*

- *if the fixed point iterations converge linearly to a simple root of f, then Aitken's method converges quadratically to the same root;*
- *if the fixed point iterations converge with order $p \geq 2$ to a simple root of f, then Aitken's method converges to the same root with order $2p - 1$;*
- *if the fixed point iterations converge linearly to a root with multiplicity $m \geq 2$ of f, then Aitken's method converges linearly to the same root with an asymptotic convergence factor of $C = 1 - 1/m$.*

In particular, if $p = 1$ and the root of f is simple, Aitken's extrapolation method converges even if the corresponding fixed point iterations diverge.

In Program 2.4 we report an implementation of Aitken's method. Here `phi` is a *function handle* associated with the expression of the iteration function of the fixed point method to which Aitken's extrapolation technique is applied. The initial datum is defined by the variable `x0`, while `tol` and `nmax` are the stopping criterion tolerance (on the absolute value of the difference between two consecutive iterates) and the maximum number of iterations allowed, respectively. If undefined, *default* values `nmax=100` and `tol=1.e-04` are assumed.

Program 2.4. aitken: Aitken method

```
function [x,niter]=aitken(phi,x0,tol,nmax,varargin)
%AITKEN Aitken's method.
% [ALPHA,NITER]=AITKEN(PHI,X0) computes an
% approximation of a fixed point ALPHA of function PHI
% starting from the initial datum X0 using Aitken's
% extrapolation method. The method stops after 100
% iterations or after the absolute value of the
% difference between two consecutive iterates is
% smaller than 1.e-04. PHI is a function handle
% associated with an anonymous function or a function
% stored in a m-file.
% [ALPHA,NITER]=AITKEN(PHI,X0,TOL,NMAX) allows to
% define the tolerance on the stopping criterion and
% the maximum number of iterations.
if nargin == 2
    tol = 1.e-04;
    nmax = 100;
elseif nargin == 3
    nmax = 100;
end
x = x0;
diff = tol + 1;
niter = 0;
while niter < nmax & diff >= tol
    gx = phi(x,varargin{:});
    ggx = phi(gx,varargin{:});
    xnew = (x*ggx-gx^2)/(ggx-2*gx+x);
    diff = abs(x-xnew);
    x = xnew;
    niter = niter  + 1;
end
if (niter==nmax & diff>tol)
    fprintf(['Fails to converge within maximum ',...
            'number of iterations\n']);
end
return
```

Example 2.11 In order to compute the single root $\alpha = 1$ for function $f(x) = e^x(x-1)$ we apply Aitken's method starting from the two following iteration functions

$$\phi_0(x) = \log(xe^x), \qquad \phi_1(x) = \frac{e^x + x}{e^x + 1}.$$

We use Program 2.4 with `tol=1.e-10`, `nmax=100`, `x0=2` and we define the two iteration functions as follows:

```
phi0 = @(x)log(x*exp(x));
phi1 = @(x)(exp(x)+x)/(exp(x)+1);
```

We now run Program 2.4 as follows:

```
[alpha,niter]=aitken(phi0,x0,tol,nmax)

alpha =
   1.0000 + 0.0000i
niter =
    10

[alpha,niter]=aitken(phi1,x0,tol,nmax)

alpha =
     1
niter =
     4
```

As we can see, the convergence is extremely rapid. For comparison the fixed point method with iteration function ϕ_1 and the same stopping criterion would have required 18 iterations, while the method corresponding to ϕ_0 would not have been convergent as $|\phi_0'(1)| = 2$. ■

Let us summarize

1. A number α satisfying $\phi(\alpha) = \alpha$ is called a fixed point of ϕ. For its computation we can use the so-called fixed point iterations: $x^{(k+1)} = \phi(x^{(k)})$;
2. fixed point iterations converge under suitable assumptions on the iteration function ϕ and its first derivative. Typically, convergence is linear, however, in the special case when $\phi'(\alpha) = 0$, the fixed point iterations converge quadratically;
3. fixed point iterations can also be used to compute the zeros of a function;
4. given a fixed point iteration $x^{(k+1)} = \phi(x^{(k)})$, it is always possible to construct a new sequence using Aitken's method, which in general converges faster.

See Exercises 2.15-2.18.

2.8 Algebraic polynomials

In this section we will consider the case where f is a polynomial of degree $n \geq 0$ of the form (1.9). As already anticipated, the space of all polynomials (1.9) is denoted by the symbol $\mathbb{P}_n$. We recall that if $p_n \in \mathbb{P}_n$, $n \geq 2$, is a polynomial whose coefficients a_k are all real, if $\alpha \in \mathbb{C}$ is a complex root of p_n, then $\bar{\alpha}$ (the complex conjugate of α) is a root of p_n too.

Abel's theorem guarantees that there does not exist an explicit form to compute all the zeros of a generic polynomial p_n, when $n \geq 5$. This fact further motivates the use of numerical methods for computing the roots of p_n.

As we have previously seen for such methods it is important to choose an appropriate initial datum $x^{(0)}$ or a suitable search interval $[a, b]$ for the root. In the case of polynomials this is sometimes possible on the basis of the following results.

Theorem 2.3 (Descartes's sign rule) *Let us denote by ν the number of sign changes of the coefficients $\{a_j\}$ and with k the number of real positive roots of a given polynomial $p_n \in \mathbb{P}_n$, each counted with its own multiplicity. Then $k \leq \nu$ and $\nu - k$ is even.*

Example 2.12 The polynomial $p_6(x) = x^6 - 2x^5 + 5x^4 - 6x^3 + 2x^2 + 8x - 8$ has zeros $\{\pm 1, \pm 2i, 1 \pm i\}$ and thus has 1 real positive root ($k = 1$). Indeed, the number of sign changes ν of its coefficients is 5 and thereafter $k \leq \nu$ and $\nu - k = 4$ is even. ■

Theorem 2.4 (Cauchy) *All of the zeros of p_n are included in the circle Γ in the complex plane*

$$\Gamma = \{z \in \mathbb{C} : \ |z| \leq 1 + \eta\}, \ \textit{where } \eta = \max_{0 \leq k \leq n-1} |a_k/a_n|. \quad (2.32)$$

This property is barely useful when $\eta \gg 1$ (for polynomial p_6 in Example 2.12 for instance, we have $\eta = 8$, while all of the roots are in circles with clearly smaller radii).

2.8.1 Hörner's algorithm

In this paragraph we will illustrate a method for the effective evaluation of a polynomial (and its derivative) in a given point z. Such algorithm allows to generate an automatic procedure, called *deflation method*, for the progressive approximation of *all* the roots of a polynomial.

From an algebraic point of view, (1.9) is equivalent to the following representation

$$p_n(x) = a_0 + x(a_1 + x(a_2 + \ldots + x(a_{n-1} + a_n x) \ldots)). \quad (2.33)$$

However, while (1.9) requires n sums and $2n - 1$ products to evaluate $p_n(x)$ (for a given x), (2.33) only requires n sums and n products. The expression (2.33), also known as the nested product algorithm, is the basis for Hörner's algorithm. This method allows to effectively evaluate the polynomial p_n in a point z by using the following *synthetic division algorithm*

$$\begin{aligned} b_n &= a_n, \\ b_k &= a_k + b_{k+1} z, \ k = n-1, n-2, ..., 0 \end{aligned} \quad (2.34)$$

In (2.34) all of the coefficients b_k with $k \leq n - 1$ depend on z and we can verify that $b_0 = p_n(z)$. The polynomial

$$q_{n-1}(x; z) = b_1 + b_2 x + ... + b_n x^{n-1} = \sum_{k=1}^{n} b_k x^{k-1}, \quad (2.35)$$

of degree $n - 1$ in x, depends on the z parameter (via the b_k coefficients) and is called the *associated polynomial* of p_n. Algorithm (2.34) is implemented in Program 2.5. The a_j coefficients of the polynomial to be evaluated are stored in vector `a` starting from a_n up to a_0.

Program 2.5. horner: synthetic division algorithm

```
function [y,b] = horner(a,z)
%HORNER Horner algorithm
%   Y=HORNER(A,Z) computes
%   Y = A(1)*Z^N + A(2)*Z^(N-1) + ... + A(N)*Z + A(N+1)
%   using Horner's synthetic division algorithm.
n = length(a)-1;
b = zeros(n+1,1);
b(1) = a(1);
for j=2:n+1
   b(j) = a(j)+b(j-1)*z;
end
y = b(n+1);
b = b(1:end-1);
return
```

We now want to introduce an effective algorithm which, knowing the root of a polynomial (or its approximation), is able to remove it and then to allow the computation of the following one until all roots are determinated.

In order to do this we should recall the following property of *polynomial division*:

Proposition 2.3 *Given two polynomials $h_n \in \mathbb{P}_n$ and $g_m \in \mathbb{P}_m$ with $m \leq n$, there are a unique polynomial $\delta \in \mathbb{P}_{n-m}$ and a unique polynomial $\rho \in \mathbb{P}_{m-1}$ such that*

$$h_n(x) = g_m(x)\delta(x) + \rho(x). \tag{2.36}$$

Thus, by dividing a polynomial $p_n \in \mathbb{P}_n$ by $x - z$, one deduces by (2.36) that

$$p_n(x) = b_0 + (x - z)q_{n-1}(x; z),$$

having denoted by q_{n-1} the quotient and by b_0 the remainder of the division. If z is a root of p_n, then we have $b_0 = p_n(z) = 0$ and therefore $p_n(x) = (x-z)q_{n-1}(x; z)$. In this case the algebric equation $q_{n-1}(x; z) = 0$ provides the $n - 1$ remaining roots of $p_n(x)$. This remark suggests to adopt the following *deflation criterion* to compute *all* the roots of p_n.

For $m = n, n - 1, \dots, 1$:

1. find a root r_m for p_m with an appropriate approximation method;
2. compute $q_{m-1}(x; r_m)$ using (2.34)-(2.35) (having set $z = r_m$);
3. set $p_{m-1} = q_{m-1}$.

In the following paragraph we propose the most widely known method in this group, which uses Newton's method for the approximation of the roots.

2.8.2 The Newton-Hörner method

As its name suggests, the *Newton-Hörner method* implements the deflation procedure using Newton's method to compute the roots r_m. The advantage lies in the fact that the implementation of Newton's method conveniently exploits Hörner's algorithm (2.34).

As a matter of fact, if q_{n-1} is the polynomial associated with p_n defined in (2.35), since

$$p_n'(x) = q_{n-1}(x; z) + (x - z)q_{n-1}'(x; z),$$

one has

$$p_n'(z) = q_{n-1}(z; z).$$

Thanks to this identity, the Newton-Hörner method for the approximation of a (real or complex) root r_j of p_n ($j = 1, \ldots, n$) takes the following form:
given an initial estimation $r_j^{(0)}$ of the root, compute for each $k \geq 0$ until convergence

$$\boxed{r_j^{(k+1)} = r_j^{(k)} - \frac{p_n(r_j^{(k)})}{p_n'(r_j^{(k)})} = r_j^{(k)} - \frac{p_n(r_j^{(k)})}{q_{n-1}(r_j^{(k)}; r_j^{(k)})}} \qquad (2.37)$$

We now use the deflation technique, exploiting the fact that $p_n(x) = (x - r_j)p_{n-1}(x)$. We can then proceed to the approximation of a zero of p_{n-1} and so on until all the roots of p_n are processed.

Consider that when $r_j \in \mathbb{C}$, it is necessary to perform the computation in complex arithmetics, taking $r_j^{(0)}$ as the non-null imaginary part. Otherwise, the Newton-Hörner method would generate a sequence $\{r_j^{(k)}\}$ of real numbers.

The Newton-Hörner method is implemented in Program 2.6. The coefficients a_j of the polynomial for which we intend to compute the roots are stored in vector `a` starting from a_n up to a_0. The other input parameters, `tol` and `nmax`, are the stopping criterion tolerance (on the absolute value of the difference between two consecutive iterates) and the maximum number of iterations allowed, respectively. If undefined, the *default* values `nmax=100` and `tol=1.e-04` are assumed. As an output, the program returns in vectors `roots` and `iter` the computed roots and the number of iterations required to compute each of the values, respectively.

Program 2.6. newtonhorner: Newton-Hörner method

```
function [roots,iter]=newtonhorner(a,x0,tol,nmax)
%NEWTONHORNER Newton-Horner method
% [ROOTS,ITER]=NEWTONHORNER(A,X0) computes the roots of
% polynomial
% P(X)= A(1)*X^N + A(2)*X^(N-1) + ... + A(N)*X + A(N+1)
% using the Newton-Horner method starting from the
% initial guess X0. The method stops for each root
% after 100 iterations or after the absolute value of
% the difference between two consecutive iterates is
% smaller than 1.e-04.
% [ROOTS,ITER]=NEWTONHORNER(A,X0,TOL,NMAX) allows to
% define the tolerance on the stopping criterion and
% the maximum number of iterations.
if nargin == 2
    tol = 1.e-04; nmax = 100;
elseif nargin == 3
    nmax = 100;
end
n=length(a)-1; roots = zeros(n,1); iter = zeros(n,1);
for k = 1:n
  % Newton iterations
  niter = 0; x = x0; diff = tol + 1;
  while niter < nmax & diff >= tol
      [pz,b] = horner(a,x);  [dpz,b] = horner(b,x);
      xnew = x - pz/dpz;         diff = abs(xnew-x);
      niter = niter + 1;         x = xnew;
  end
  if (niter==nmax & diff> tol)
      fprintf([' Fails to converge within maximum ',...
               'number of iterations\n ']);
  end
  % Deflation
  [pz,a] = horner(a,x); roots(k) = x; iter(k) = niter;
end
```

Remark 2.2 In order to minimize the propagation of roundoff errors, during the deflation process it is better to first approximate the root r_1 with smallest absolute value and then to proceed to the computation of the following roots $r_2, r_3, \ldots$, until the one with the largest absolute value is reached (to learn more, see for instance [QSS07]). ■

Example 2.13 To compute the roots $\{1, 2, 3\}$ of the polynomial $p_3(x) = x^3 - 6x^2 + 11x - 6$ we use Program 2.6

```
a=[1 -6 11 -6]; [x,niter]=newtonhorner(a,0,1.e-15,100)
x =
     1
     2
     3
niter =
     8
     8
     2
```

The method computes all three roots accurately and in few iterations. As pointed out in Remark 2.2 however, the method is not always so effective. For instance, if we consider the polynomial $p_4(x) = x^4 - 7x^3 + 15x^2 - 13x + 4$ (which has the root 1 of multiplicity 3 and a single root with value 4) we find the following results

```
a=[1 -7 15 -13 4]; format long;
[x,niter]=newtonhorner(a,0,1.e-15,100)

x =
   1.000006935337374
   0.999972452635761
   1.000020612232168
   3.999999999794697

niter =
    61
   100
     6
     2
```

The loss of accuracy is quite evident for the computation of the multiple root, and becomes as more relevant as the multiplicity increases. More in general, it can be shown (see [QSS07]) that the problem of root-finding for a function f becomes ill-conditioned (that is, very sensitive to perturbations on the data) as the derivative f' gets small at the roots. For an instance, see Exercise 2.6.

■

2.9 What we haven't told you

The most sophisticated methods for the computation of the zeros of a function combine different algorithms. In particular, the MATLAB function `fzero` (see Section 1.5.1) adopts the so called Dekker-Brent method (see [QSS07], Section 6.2.3). In its basic form `fzero(fun,x0)` computes the zero of the function associated with the function handle `fun`, starting from `x0`.

For instance, we could solve the problem in Example 2.1 also by `fzero`, using the initial value `x0=0.3` (as done by Newton's method) via the following instructions:

```
M=6000; v=1000; f=@(r) M-v*(1+r)/r*((1+r)^5-1);
x0=0.3;
[alpha,res,flag,info]=fzero(f,x0);
```

We find the root `alpha=0.06140241153653` after 7 iterations and 29 evaluations of the function `f`, with a residual `res=-1.8190e-12`.
The variable `info` is a *structure* with 5 subfields. Precisely the fields `info.iterations` and `info.funcCount` contain number of iterations and global number of function evaluations performed during the call, respectively. We note that when output parameter `flag` assumes a negative occurrence, then the function `fzero` failed in searching the zero.

For a comparison, Newton method converges in 6 iterations to the value 0.06140241153653 with a residual equal to 9.0949e-13, but it requires the knowledge of the first derivative of f and a total of 12 function evaluations.

In order to compute the zeros of a polynomial, in addition to the Newton-Hörner method, we can cite the methods based on Sturm sequences, Müller's method, (see [Atk89] or [QSS07]) and Bairstow's method ([RR01], page 371 and following). A different approach consists in characterizing the zeros of a function as the eigenvalues of a special matrix (called the *companion matrix*) and then using appropriate techniques for their computation. This approach is adopted by the MATLAB function `roots` which has been introduced in Section 1.5.2.

We have mentioned in Section 2.5 how to set up a Newton method for a nonlinear system, like (2.15). More in general, any fixed point iteration can be easily extended to compute the roots of nonlinear systems. Other methods exist as well, such as the Broyden and quasi-Newton methods, which can be regarded as generalizations of Newton's method (see [JS96], [Deu04], [SM03] and [QSS07, Chapter 7]).

The MATLAB instruction

```
zero=fsolve(@fun,x0)
```

`fsolve`

allows the computation of one zero of a nonlinear system defined via the user-defined function `fun` starting from the vector `x0` as initial guess. The function `fun` returns the n values $f_i(\bar{x}_1,\ldots,\bar{x}_n)$, $i = 1,\ldots,n$, for any given input vector $(\bar{x}_1,\ldots,\bar{x}_n)^T$.

For instance, in order to solve the nonlinear system (2.17) using `fsolve` the corresponding user-defined function, which we call `systemnl`, is defined as follows:

```
function fx=systemnl(x)
fx(1) = x(1)^2+x(2)^2-1;
fx(2) = sin(pi*0.5*x(1))+x(2)^3;
```

The MATLAB instructions to solve this system are therefore:

```
x0 = [1 1];
alpha=fsolve(@systemnl,x0)

alpha =
     0.4761    -0.8794
```

Using this procedure we have found only one of the two roots. The other can be computed starting from the initial datum `-x0`.

Octave 2.1 The commands `fzero` and `fsolve` have exactly the same purpose in MATLAB and Octave, however their interface differ slightly in what concerns the optional arguments. We encourage the reader to study the `help` documentation of both commands in each environment. ■

2.10 Exercises

Exercise 2.1 Given the function $f(x) = \cosh x + \cos x - \gamma$, for $\gamma = 1, 2, 3$ find an interval that contains the zero of f. Then compute the zero by the bisection method with a tolerance of 10^{-10}.

Exercise 2.2 (State equation of a gas) For carbon dioxide (CO_2) the coefficients a and b in (2.1) take the following values: $a = 0.401\text{Pa m}^6$, $b = 42.7 \cdot 10^{-6}\text{m}^3$ (Pa stands for Pascal). Find the volume occupied by 1000 molecules of CO_2 at a temperature $T = 300\text{K}$ and a pressure $p = 3.5 \cdot 10^7$ Pa by the bisection method, with a tolerance of 10^{-12} (the Boltzmann constant is $k = 1.3806503 \cdot 10^{-23}$ Joule K^{-1}).

Exercise 2.3 Consider a plane whose slope varies with constant rate ω, and a dimensionless object which is steady at the initial time $t = 0$. At time $t > 0$ its position is

$$s(t, \omega) = \frac{g}{2\omega^2}[\sinh(\omega t) - \sin(\omega t)],$$

where $g = 9.8 \text{ m/s}^2$ denotes the gravity acceleration. Assuming that this object has moved by 1 meter in 1 second, compute the corresponding value of ω with a tolerance of 10^{-5}.

Exercise 2.4 Prove inequality (2.6).

Exercise 2.5 Motivate why in Program 2.1 the instruction `x(2) = x(1)+(x(3)- x(1))*0.5` has been used instead of the more natural one `x(2)=(x(1)+x(3))*0.5` in order to compute the midpoint.

Exercise 2.6 Apply Newton's method to solve Exercise 2.1. Why is this method not accurate when $\gamma = 2$?

Exercise 2.7 Apply Newton's method to compute the square root of a positive number a. Proceed in a similar manner to compute the cube root of a.

Exercise 2.8 Assuming that Newton's method converges, show that (2.9) is true when α is a simple root of $f(x) = 0$ and f is twice continuously differentiable in a neighborhood of α.

Exercise 2.9 (Rods system) Apply Newton's method to solve Problem 2.3 for $\beta \in [0, 2\pi/3]$ with a tolerance of 10^{-5}. Assume that the lengths of the rods are $a_1 = 10$ cm, $a_2 = 13$ cm, $a_3 = 8$ cm and $a_4 = 10$ cm. For each value of β consider two possible initial data, $x^{(0)} = -0.1$ and $x^{(0)} = 2\pi/3$.

Exercise 2.10 Notice that the function $f(x) = e^x - 2x^2$ has 3 zeros, $\alpha_1 < 0$, α_2 and α_3 positive. For which value of $x^{(0)}$ does Newton's method converge to α_1?

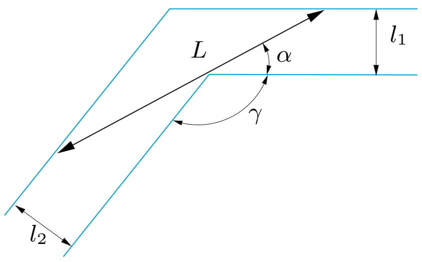


Figure 2.11. The problem of a rod sliding in a corridor

Exercise 2.11 Use Newton's method to compute the zero of $f(x) = x^3 - 3x^2 2^{-x} + 3x4^{-x} - 8^{-x}$ in $[0, 1]$ and explain why convergence is not quadratic.

Exercise 2.12 A projectile is ejected with velocity v_0 and angle α in a tunnel of height h and reaches its maximum range when α is such that $\sin(\alpha) = \sqrt{2gh/v_0^2}$, where $g = 9.8$ m/s^2 is the gravity acceleration. Compute α using Newton's method, assuming that $v_0 = 10$ m/s and $h = 1$ m.

Exercise 2.13 (Investment fund) Solve Problem 2.1 by Newton's method with a tolerance of 10^{-12}, assuming $M = 6000$ euros, $v = 1000$ euros and $n = 5$. As an initial guess take the result obtained after 5 iterations of the bisection method applied on the interval $(0.01, 0.1)$.

Exercise 2.14 A corridor has the form indicated in Figure 2.11. The maximum length L of a rod that can pass from one extreme to the other by sliding on the ground is given by

$$L = l_2/(\sin(\pi - \gamma - \alpha)) + l_1/\sin(\alpha),$$

where α is the solution of the nonlinear equation

$$l_2 \frac{\cos(\pi - \gamma - \alpha)}{\sin^2(\pi - \gamma - \alpha)} - l_1 \frac{\cos(\alpha)}{\sin^2(\alpha)} = 0. \qquad (2.38)$$

Compute α by Newton's method when $l_2 = 10$, $l_1 = 8$ and $\gamma = 3\pi/5$.

Exercise 2.15 Let ϕ_N be the iteration function of Newton's method when regarded as a fixed point iteration. Show that $\phi_N'(\alpha) = 1 - 1/m$ where α is a zero of f with multiplicity m. Deduce that Newton's method converges quadratically if α is a simple root of $f(x) = 0$, and linearly otherwise.

Exercise 2.16 Deduce from the graph of $f(x) = x^3 + 4x^2 - 10$ that this function has a unique real zero α. To compute α use the following fixed point iterations: given $x^{(0)}$, define $x^{(k+1)}$ such that

$$x^{(k+1)} = \frac{2(x^{(k)})^3 + 4(x^{(k)})^2 + 10}{3(x^{(k)})^2 + 8x^{(k)}}, \qquad k \geq 0$$

and analyze its convergence to α.

Exercise 2.17 Analyze the convergence of the fixed point iterations

$$x^{(k+1)} = \frac{x^{(k)}[(x^{(k)})^2 + 3a]}{3(x^{(k)})^2 + a}, \quad k \geq 0,$$

for the computation of the square root of a positive number a.

Exercise 2.18 Repeat the computations carried out in Exercise 2.11 this time using the stopping criterion based on the residual. Which result is the more accurate?

3

Approximation of functions and data

Approximating a function f consists of replacing it by another function $\tilde{f}$ of simpler form that may be used as its surrogate. This strategy is used frequently in numerical integration where, instead of computing $\int_a^b f(x)dx$, one carries out the exact computation of $\int_a^b \tilde{f}(x)dx$, $\tilde{f}$ being a function simple to integrate (e.g. a polynomial), as we will see in the next chapter. In other instances the function f may be available only partially through its values at some selected points. In these cases we aim at constructing a continuous function $\tilde{f}$ that could represent the empirical law which is behind the finite set of data. We provide some examples which illustrate this kind of approach.

3.1 Some representative problems

Problem 3.1 (Climatology) The air temperature near the ground depends on the concentration K of the carbon acid (H_2CO_3) therein. In Table 3.1 (taken from Philosophical Magazine 41, 237 (1896)) we report for different latitudes on the Earth and for four different values of K, the variation $\delta_K = \theta_K - \theta_{\bar{K}}$ of the average temperature with respect to the average temperature corresponding to a reference value $\bar{K}$ of K. Here $\bar{K}$ refers to the value measured in 1896, and is normalized to one. In this case we can generate a function that, on the basis of the available data, provides an approximate value of the average temperature at any possible latitude and for other values of K (see Example 3.1). ■

Problem 3.2 (Finance) In Figure 3.1 we report the price of a stock at the Zurich stock exchange over two years. The curve was obtained by joining with a straight line the prices reported at every day's closure. This simple representation indeed implicitly assumes that the prices change linearly in the course of the day (we anticipate that this approximation

Table 3.1. Variation of the average yearly temperature on the Earth for four different values of the concentration K of carbon acid at different latitudes

	δ_K			
Latitude	$K = 0.67$	$K = 1.5$	$K = 2.0$	$K = 3.0$
65	-3.1	3.52	6.05	9.3
55	-3.22	3.62	6.02	9.3
45	-3.3	3.65	5.92	9.17
35	-3.32	3.52	5.7	8.82
25	-3.17	3.47	5.3	8.1
15	-3.07	3.25	5.02	7.52
5	-3.02	3.15	4.95	7.3
-5	-3.02	3.15	4.97	7.35
-15	-3.12	3.2	5.07	7.62
-25	-3.2	3.27	5.35	8.22
-35	-3.35	3.52	5.62	8.8
-45	-3.37	3.7	5.95	9.25
-55	-3.25	3.7	6.1	9.5

is called composite linear interpolation). We ask whether from this graph one could predict the stock price for a short time interval beyond the time of the last quotation. We will see in Section 3.6 that this kind of prediction could be guessed by resorting to a special technique known as *least-squares* approximation of data (see Example 3.12). ■

Problem 3.3 (Biomechanics) We consider a mechanical test to establish the link between stresses and deformations of a sample of biological tissue (an intervertebral disc, see Figure 3.2). Starting from the data collected in Table 3.2 (taken from P.Komarek, Chapt. 2 of *Biomechanics of Clinical Aspects of Biomedicine*, 1993, J.Valenta ed., Elsevier) in

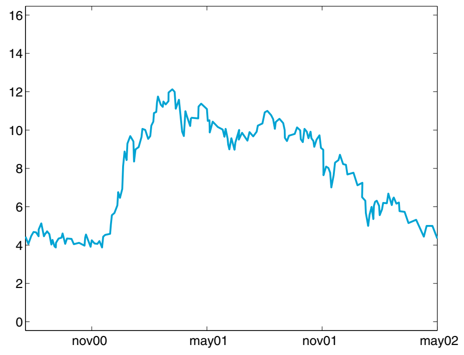


Figure 3.1. Price variation of a stock over two years

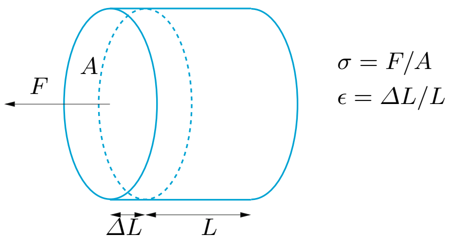


Figure 3.2. A schematic representation of an intervertebral disc

Table 3.2. Values of the deformation for different values of a stress applied on an intervertebral disc

test	stress σ	stress ϵ	test	stress σ	stress ϵ
1	0.00	0.00	5	0.31	0.23
2	0.06	0.08	6	0.47	0.25
3	0.14	0.14	7	0.60	0.28
4	0.25	0.20	8	0.70	0.29

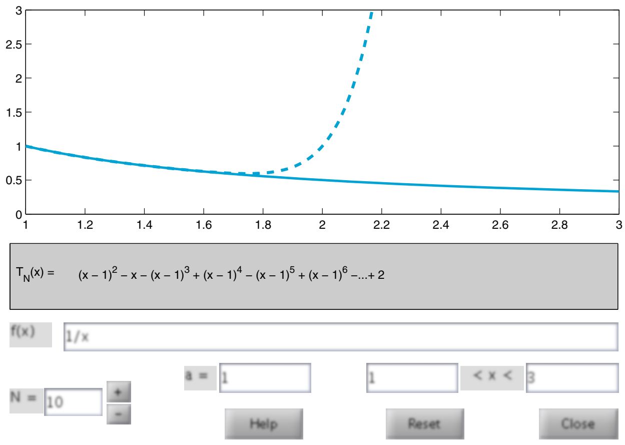
Example 3.13 we will estimate the deformation corresponding to a stress $\sigma = 0.9$ MPa (MPa= 100 N/cm^2). ■

Problem 3.4 (Robotics) We want to approximate the planar trajectory followed by a robot (idealized as a material point) during a working cycle in an industry. The robot should satisfy a few constraints: it must be steady at the point $(0, 0)$ in the plane at the initial time (say, $t = 0$), transit through the point $(1, 2)$ at $t = 1$, get the point $(4, 4)$ at $t = 2$, stop and restart immediately and reach the point $(3, 1)$ at $t = 3$, return to the initial point at time $t = 5$, stop and restart a new working cycle. In Example 3.10 we will solve this problem using the *splines* functions. ■

3.2 Approximation by Taylor's polynomials

A function f in a given interval can be replaced by its Taylor polynomial, which was introduced in Section 1.5.3. This technique is computationally expensive since it requires the knowledge of f and its derivatives up to the order n (the polynomial degree) at a given point x_0. Moreover, the Taylor polynomial may fail to accurately represent f far enough from the point x_0. For instance, in Figure 3.3 we compare the behavior of $f(x) = 1/x$ with that of its Taylor polynomial of degree 10 built around the point $x_0 = 1$. This picture also shows the graphical interface of the MATLAB function `taylortool` which allows the computation of Taylor's polynomial of arbitrary degree for any given function f. The

`taylortool`



$T_N(x) = (x-1)^2 - x - (x-1)^3 + (x-1)^4 - (x-1)^5 + (x-1)^6 - \ldots + 2$

f(x) 1/x

N = 10   a = 1   1 < x < 3

Figure 3.3. Comparison between the function $f(x) = 1/x$ (*solid line*) and its Taylor polynomial of degree 10 related to the point $x_0 = 1$ (*dashed line*). The explicit form of the Taylor polynomial is also reported

agreement between the function and its Taylor polynomial is very good in a small neighborhood of $x_0 = 1$ while it becomes unsatisfactory when $x - x_0$ gets large. Fortunately, this is not the case of other functions such as the exponential function which is approximated quite nicely for all $x \in \mathbb{R}$ by its Taylor polynomial related to $x_0 = 0$, provided that the degree n is sufficiently large.

In the course of this chapter we will introduce approximation methods that are based on alternative approaches.

Octave 3.1 `taylortool` is not available in Octave. ■

3.3 Interpolation

As seen in Problems 3.1, 3.2 and 3.3, in several applications it may happen that a function is known only through its values at some given points. We are therefore facing a (general) case where $n + 1$ couples $\{x_i, y_i\}$, $i = 0, \ldots, n$, are given; the points x_i are all distinct and are called *nodes*.

For instance in the case of Table 3.1, n is equal to 12, the nodes x_i are the values of the latitude reported in the first column, while the y_i are the corresponding values (of the temperature variation) in the remaining columns.

In such a situation it seems natural to require the approximate function $\tilde{f}$ to satisfy the set of relations

$$\boxed{\tilde{f}(x_i) = y_i,\ i = 0, 1, \ldots, n} \tag{3.1}$$

Such an $\tilde{f}$ is called *interpolant* of the set of data $\{y_i\}$ and equations (3.1) are the interpolation conditions.

Several kinds of interpolants could be envisaged, such as:

- *polynomial interpolant*:

$$\tilde{f}(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n;$$

- *trigonometric interpolant*:

$$\tilde{f}(x) = a_{-M} e^{-iMx} + \ldots + a_0 + \ldots + a_M e^{iMx}$$

  where M is an integer equal to $n/2$ if n is even, $(n+1)/2$ if n is odd, and i is the imaginary unit;
- *rational interpolant*:

$$\tilde{f}(x) = \frac{a_0 + a_1 x + \ldots + a_k x^k}{a_{k+1} + a_{k+2} x + \ldots + a_{k+n+1} x^n}.$$

For simplicity we only consider those interpolants which depend linearly on the unknown coefficients a_i. Both polynomial and trigonometric interpolation fall into this category, whereas the rational interpolant does not.

3.3.1 Lagrangian polynomial interpolation

Let us focus on the polynomial interpolation. The following result holds:

Proposition 3.1 *For any set of couples $\{x_i, y_i\}$, $i = 0, \ldots, n$, with distinct nodes x_i, there exists a unique polynomial of degree less than or equal to n, which we indicate by Π_n and call interpolating polynomial of the values y_i at the nodes x_i, such that*

$$\boxed{\Pi_n(x_i) = y_i,\ i = 0, \ldots, n} \tag{3.2}$$

In the case where the $\{y_i,\ i = 0, \ldots, n\}$ represent the values of a continuous function f, Π_n is called interpolating polynomial of f (in short, interpolant of f) and will be denoted by $\Pi_n f$.

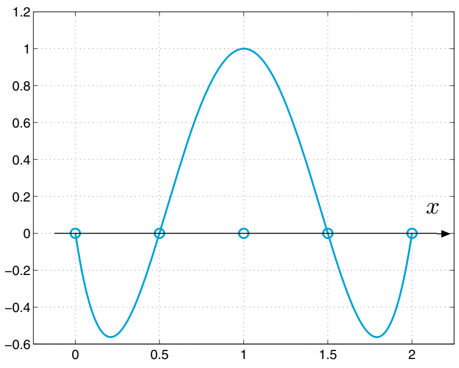


Figure 3.4. The polynomial $\varphi_2 \in \mathbb{P}_4$ associated with a set of 5 equispaced nodes

To verify uniqueness we proceed by contradiction and suppose that there exist two distinct polynomials of degree n, Π_n and Π_n^*, both satisfying the nodal relation (3.2). Their difference, $\Pi_n - \Pi_n^*$, would be a polynomial of degree n which vanishes at $n+1$ distinct points. Owing to a well known theorem of Algebra, such a polynomial should vanish identically, and then Π_n^* must coincide with Π_n.

In order to obtain an expression for Π_n, we start from a very special case where y_i vanishes for all i apart from $i = k$ (for a fixed k) for which $y_k = 1$. Then setting $\varphi_k(x) = \Pi_n(x)$, we must have (see Figure 3.4)

$$\varphi_k \in \mathbb{P}_n,\ \varphi_k(x_j) = \delta_{jk} = \begin{cases} 1 & \text{if } j = k, \\ 0 & \text{otherwise,} \end{cases}$$

where δ_{jk} is the Kronecker symbol.

The functions φ_k have the following expression:

$$\varphi_k(x) = \prod_{\substack{j=0 \\ j\neq k}}^{n} \frac{x - x_j}{x_k - x_j}, \qquad k = 0, \ldots, n. \tag{3.3}$$

We move now to the general case where $\{y_i, i = 0, \ldots, n\}$ is a set of arbitrary values. Using an obvious superposition principle we can obtain the following expression for Π_n

$$\boxed{\Pi_n(x) = \sum_{k=0}^{n} y_k \varphi_k(x)} \tag{3.4}$$

Indeed, this polynomial satisfies the interpolation conditions (3.2), since

$$\Pi_n(x_i) = \sum_{k=0}^{n} y_k \varphi_k(x_i) = \sum_{k=0}^{n} y_k \delta_{ik} = y_i, \quad i = 0, \ldots, n.$$

Due to their special role, the functions φ_k are called *Lagrange characteristic polynomials*, and (3.4) is the *Lagrange form* of the interpolant.

In MATLAB we can store the `n+1` couples $\{(x_i, y_i)\}$ in the vectors `x` and `y`, and then the instruction `c=polyfit(x,y,n)` will provide the coefficients of the interpolating polynomial. Precisely, `c(1)` will contain the coefficient of x^n, `c(2)` that of x^{n-1}, ... and `c(n+1)` the value of $\Pi_n(0)$. (More on this command can be found in Section 3.6.) As already seen in Chapter 1, we can then use the instruction `p=polyval(c,z)` to compute the value `p(j)` attained by the interpolating polynomial at `z(j)`, `j=1,...,m`, the latter being a set of `m` arbitrary points.

polyfit

In the case when the explicit form of the function `f` is available, the Lagrange interpolant of f will be denoted by $\Pi_n f$. In order to obtain the vector `y` of values of `f` at some specific nodes (which should be stored in a vector `x`) we can use the instruction `y=f(x)`.

Example 3.1 (Climatology) To obtain the interpolating polynomial for the data of Problem 3.1 relating to the value $K = 0.67$ (first column of Table 3.1), using only the values of the temperature for the latitudes 65, 35, 5, -25, -55, we can use the following MATLAB instructions:

```
x=[-55 -25 5 35 65]; y=[-3.25 -3.2 -3.02 -3.32 -3.1];
format short e; c=polyfit(x,y,4)
```

```
 c =
   8.2819e-08  -4.5267e-07  -3.4684e-04   3.7757e-04  -3.0132e+00
```

The graph of the interpolating polynomial can be obtained as follows:

```
z=linspace(x(1),x(end),100);
p=polyval(c,z);
plot(z,p,x,y,'o');grid on;
```

In order to obtain a smooth curve we have evaluated our polynomial at 101 equispaced points in the interval $[-55, 65]$ (as a matter of fact, MATLAB plots are always constructed on piecewise linear interpolation between neighboring points). Note that the instruction `x(end)` picks up directly the last component of the vector `x`, without specifying the length of the vector. In Figure 3.5 the filled circles correspond to those values which have been used to construct the interpolating polynomial, whereas the empty circles correspond to values that have not been used. We can appreciate the qualitative agreement between the curve and the data distribution. ■

Using the following result we can evaluate the error obtained by replacing f with its interpolating polynomial $\Pi_n f$:

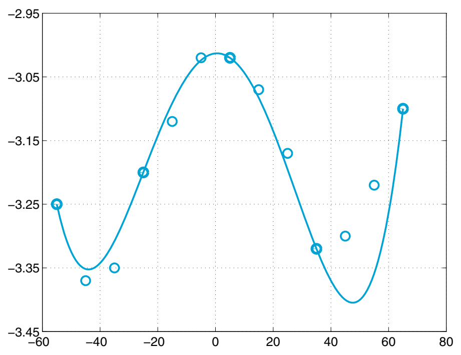


Figure 3.5. The interpolating polynomial of degree 4 introduced in Example 3.1

Proposition 3.2 *Let I be a bounded interval, and consider $n+1$ distinct interpolation nodes $\{x_i, i=0,\dots,n\}$ in I. Let f be continuously differentiable up to order $n+1$ in I. Then $\forall x \in I$ $\exists \xi_x \in I$ such that*

$$\boxed{E_n f(x) = f(x) - \Pi_n f(x) = \frac{f^{(n+1)}(\xi_x)}{(n+1)!}\prod_{i=0}^{n}(x-x_i)} \tag{3.5}$$

Obviously, $E_n f(x_i) = 0$, $i = 0, \dots, n$. Result (3.5) can be better specified in the case of a uniform distribution of nodes, that is when $x_i = x_{i-1} + h$ for $i = 1, \dots, n$, for a given $h > 0$ and a given x_0. As stated in Exercise 3.1, $\forall x \in (x_0, x_n)$ one can verify that

$$\left|\prod_{i=0}^{n}(x-x_i)\right| \le n!\frac{h^{n+1}}{4}, \tag{3.6}$$

and therefore

$$\max_{x\in I}|E_n f(x)| \le \frac{\max\limits_{x\in I}|f^{(n+1)}(x)|}{4(n+1)}h^{n+1}. \tag{3.7}$$

Unfortunately, we cannot deduce from (3.7) that the error tends to 0 when $n \to \infty$, in spite of the fact that $h^{n+1}/[4(n+1)]$ tends to 0. In fact, as shown in Example 3.2, there exist functions f for which the limit can even be infinite, that is

$$\lim_{n\to\infty}\max_{x\in I}|E_n f(x)| = \infty.$$

This striking result indicates that by increasing the degree n of the interpolating polynomial we do not necessarily obtain a better reconstruction of f. For instance, should we use all data of the second column of Table 3.1, we would obtain the interpolating polynomial $\Pi_{12}f$ represented in Figure 3.6, left, whose behavior in the vicinity of the left-hand of the interval is far less satisfactory than that obtained in Figure 3.5 using a much smaller number of nodes. An even worse result may arise for a special class of functions, as we report in the next example.

Example 3.2 (Runge) If the function $f(x) = 1/(1 + x^2)$ is interpolated at equispaced nodes in the interval $I = [-5, 5]$, the error $\max_{x\in I} |E_n f(x)|$ tends to infinity when $n \to \infty$. This is due to the fact that if $n \to \infty$ the order of magnitude of $\max_{x\in I} |f^{(n+1)}(x)|$ outweighs the infinitesimal order of $h^{n+1}/[4(n+1)]$. This conclusion can be verified by computing the maximum of f and its derivatives up to the order 21 by means of the following MATLAB instructions:

```
syms x; n=20; f=1/(1+x^2);  df=diff(f,1);
cdf=matlabFunction(df);
for i = 1:n+1
 df = diff(df,1); cdfn=matlabFunction(df);
 x = fzero(cdfn,0); M(i) = abs(cdf(x)); cdf = cdfn;
end
```

The maximum of the absolute values of the functions $f^{(n)}$, $n = 1, \ldots, 21$, are stored in the vector M. Notice that the command matlabFunction converts the symbolic expression df into a function handle that is passed to the function fzero. In particular, the absolute values of $f^{(n)}$ for $n = 3,\ 9,\ 15,\ 21$ are:

matlab-Function

```
format short e; M([3,9,15,21])

ans =
   4.6686e+00   3.2426e+05   1.2160e+12   4.8421e+19
```

while the corresponding values of the maximum of $\prod_{i=0}^{n}(x - x_i)/(n + 1)!$ are

```
z = linspace(-5,5,10000);
for n=0:20; h=10/(n+1); x=[-5:h:5];
  c=poly(x); r(n+1)=max(polyval(c,z));
  r(n+1)=r(n+1)/prod([1:n+1]);
end
r([3,9,15,21])

ans =
   1.1574e+01   5.1814e-02   1.3739e-05   4.7247e-10
```

where c=poly(x) is a vector whose components are the coefficients of that polynomial whose roots are the elements of the vector x. It follows that $\max_{x\in I} |E_n f(x)|$ attains the following values:

poly

```
   5.4034e+01   1.6801e+04   1.6706e+07   2.2877e+10
```

for $n = 3,\ 9,\ 15,\ 21$, respectively. The lack of convergence is also indicated by the presence of severe oscillations in the graph of the interpolating polynomial with respect to the graph of f, especially near the endpoints of the interval (see Figure 3.6, right). This behavior is known as *Runge's phenomenon*. ■

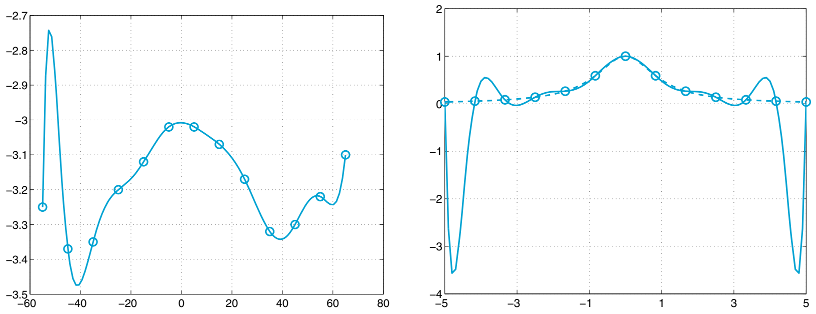


Figure 3.6. Two examples of Runge's phenomenon: at left, Π_{12} computed for the data of Table 3.1, column $K = 0.67$; at right, $\Pi_{12} f$ (*solid line*) computed on 13 equispaced nodes for the function $f(x) = 1/(1 + x^2)$ (*dashed line*)

Besides (3.7), the following inequality can also be proved:

$$\max_{x\in I} |f'(x) - (\Pi_n f)'(x)| \leq Ch^n \max_{x\in I} |f^{(n+1)}(x)|,$$

where C is a constant independent of h. Therefore, if we approximate the first derivative of f by the first derivative of $\Pi_n f$, we loose an order of convergence with respect to h.

polyder In MATLAB, $(\Pi_n f)'$ can be computed using the instruction `[d]= polyder(c)`, where `c` is the input vector in which we store the coefficients of the interpolating polynomial, while `d` is the output vector where we store the coefficients of its first derivative (see Section 1.5.2).

Octave 3.2 The command `matlabFunction` is not available in Octave. ■

3.3.2 Stability of polynomial interpolation

What happens to the interpolating polynomials if, instead of considering exact values $f(x_i)$ we consider perturbed ones, say $\hat{f}(x_i)$, with $i = 0, \ldots, n$? Note that perturbations arise because of either rounding errors or uncertainty in measuring data themselves.

Let $\Pi_n \hat{f}$ be the exact polynomial interpolating the values $\hat{f}(x_i)$. Denoting by $\mathbf{x}$ the vector whose components are the interpolation nodes $\{x_i\}$, we have

$$\begin{aligned}\max_{x\in I} |\Pi_n f(x) - \Pi_n \hat{f}(x)| &= \max_{x\in I} \left| \sum_{i=0}^{n} \left(f(x_i) - \hat{f}(x_i) \right) \varphi_i(x) \right| \\ &\leq \Lambda_n(\mathbf{x}) \max_{0\leq i\leq n} \left| f(x_i) - \hat{f}(x_i) \right|\end{aligned} \tag{3.8}$$

where

$$\Lambda_n(\mathbf{x}) = \max_{x \in I} \sum_{i=0}^{n} |\varphi_i(x)| \tag{3.9}$$

is the so-called *Lebesgue's constant* which depends on interpolation nodes. Small variations on the nodal values $f(x_i)$ yield small changes on the interpolating polynomial, provided that the Lebesgue's constant is small. Λ_n can therefore be regarded as a *condition number* of the interpolation problem. For Lagrange interpolation at equispaced nodes one has

$$\Lambda_n(\mathbf{x}) \simeq \frac{2^{n+1}}{en(\log n + \gamma)}, \tag{3.10}$$

where $e \simeq 2.71834$ is the Napier (or Euler) number, while $\gamma \simeq 0.547721$ is the Euler constant (see [Hes98] and [Nat65]).

For large values of n, Lagrange interpolation on equispaced nodes can therefore be unstable, as we can deduce from the following example. (See also the Exercise 3.8.)

Example 3.3 To interpolate $f(x) = \sin(2\pi x)$ at 22 equispaced nodes in the interval $[-1, 1]$, let us generate the values $\hat{f}(x_i)$ by a random perturbation of the exact values $f(x_i)$, such that

$$\max_{i=0,\ldots,21} |f(x_i) - \hat{f}(x_i)| \simeq 9.5 \cdot 10^{-4}.$$

In Figure 3.7 the two interpolating polynomials $\Pi_{21} f$ and $\Pi_{21} \hat{f}$ are compared, the difference between the two polynomials is much larger than the perturbations on data, precisely $\max_{x \in I} |\Pi_n f(x) - \Pi_n \hat{f}(x)| \simeq 3.1342$, and the gap is especially severe near the endpoints of the interval. Note that in this example the Lebesgue's constant is very high, being $\Lambda_{21}(\mathbf{x}) \simeq 20454$. ■

See the Exercises 3.1-3.4.

3.3.3 Interpolation at Chebyshev nodes

Runge's phenomenon can be avoided if a suitable distribution of nodes is used. In particular, in an arbitrary interval $[a, b]$, we can consider the so called *Chebyshev-Gauss-Lobatto nodes* (see Figure 3.8, right):

$$\boxed{x_i = \frac{a+b}{2} + \frac{b-a}{2}\widehat{x}_i, \text{ where } \widehat{x}_i = -\cos(\pi i/n),\ i = 0, \ldots, n} \tag{3.11}$$

Obviously, $x_i = \widehat{x}_i$, $i = 0, \ldots, n$, when $[a, b] = [-1, 1]$. Indeed, for this special distribution of nodes it is possible to prove that, if f is a continuous and differentiable function in $[a, b]$, $\Pi_n f$ converges to f as $n \to \infty$ for all $x \in [a, b]$.

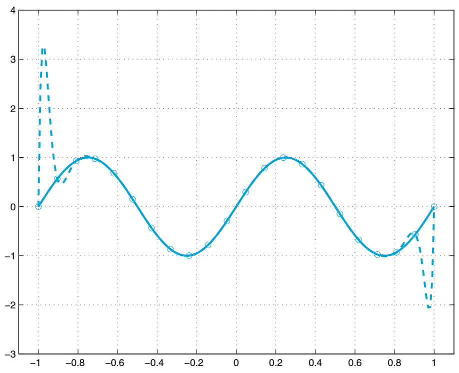


Figure 3.7. The effect of instability on equispaced Lagrange interpolation. $\Pi_{21}f$ (*solid line*) and $\Pi_{21}\hat{f}$ (*dashed line*) represent the exact and perturbed interpolation polynomials, respectively, for the Example 3.3

The Chebyshev-Gauss-Lobatto nodes, which are the abscissas of equispaced nodes on the unit semi-circumference, lie inside $[a, b]$ and are clustered near the endpoints of this interval (see Figure 3.8, right).

Another nonuniform distribution of nodes in the interval (a, b), sharing the same convergence properties is provided by *t*he Chebyshev-Gauss nodes:

$$\boxed{x_i = \frac{a+b}{2} - \frac{b-a}{2}\cos\left(\frac{2i+1}{n+1}\frac{\pi}{2}\right),\ i = 0, \ldots, n} \tag{3.12}$$

Example 3.4 We consider anew the function f of Runge's example and compute its interpolating polynomial at Chebyshev-Gauss-Lobatto nodes. The latter can be obtained through the following MATLAB instructions:

```
xc = -cos(pi*[0:n]/n); x = (a+b)*0.5+(b-a)*xc*0.5;
```

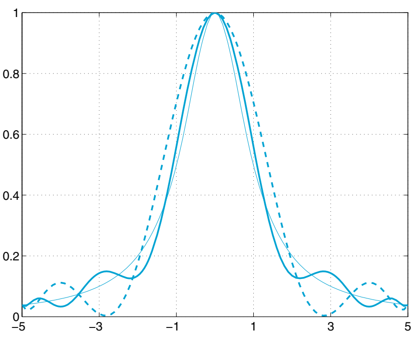
where `n+1` is the number of nodes, while `a` and `b` are the endpoints of the interpolation interval (in the sequel we choose `a=-5` and `b=5`). Then we compute the interpolating polynomial by the following instructions:

```
f= @(x) 1./(1+x.^2); y = f(x); c = polyfit(x,y,n);
```

Now let us compute the absolute values of the differences between f and its interpolant relative to Chebyshev-Gauss-Lobatto nodes at as many as 1000 equispaced points in the interval $[-5, 5]$ and take the maximum error values:

```
x1 = linspace(-5,5,1000); p=polyval(c,x1);
f1 = f(x1); err = max(abs(p-f1));
```

As we see in Table 3.3, the maximum of the error decreases when n increases. ■



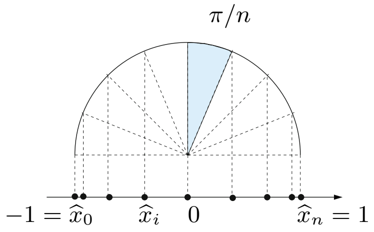


Figure 3.8. The left side picture shows the comparison between the function $f(x) = 1/(1+x^2)$ (*thin solid line*) and its interpolating polynomials of degree 8 (*dashed line*) and 12 (*solid line*) at the Chebyshev-Gauss-Lobatto nodes. Note that the amplitude of spurious oscillations decreases as the degree increases. The right side picture shows the distribution of Chebyshev-Gauss-Lobatto nodes in the interval $[-1, 1]$

Table 3.3. The interpolation error for Runge's function $f(x) = 1/(1+x^2)$ when the Chebyshev-Gauss-Lobatto nodes (3.11) are used

n	5	10	20	40
E_n	0.6386	0.1322	0.0177	0.0003

When the Lagrange interpolant is defined at the Chebyshev-Gauss-Lobatto nodes (3.11), then the Lebesgue's constant can be bounded as follows ([Hes98])

$$\Lambda_n(\mathbf{x}) < \frac{2}{\pi}\left(\log n + \gamma + \log\frac{8}{\pi}\right) + \frac{\pi}{72\ n^2}, \tag{3.13}$$

while when interpolation is carried out on the Chebyshev-Gauss nodes (3.12), then

$$\Lambda_n(\mathbf{x}) < \frac{2}{\pi}\left(\log(n+1) + \gamma + \log\frac{8}{\pi}\right) + \frac{\pi}{72(n+1)^2}. \tag{3.14}$$

As usual, $\gamma \simeq 0.57721$ denotes the Euler constant.

By comparing (3.13) and (3.14) with the estimate (3.10), we can conclude that the Lagrange interpolation at Chebyshev nodes is much less sensitive to perturbation errors than interpolation at equispaced nodes.

Example 3.5 Let us use now interpolation at the Chebyshev nodes, either (3.11) and (3.12). Starting from the same data perturbations considered in Example 3.3, when $n = 21$ we have $\max\limits_{x\in I}|\Pi_n f(x) - \Pi_n \hat{f}(x)| \simeq 1.0977\cdot 10^{-3}$ for nodes (3.11), while $\max_{x\in I}|\Pi_n f(x) - \Pi_n \hat{f}(x)| \simeq 1.1052 \cdot 10^{-3}$ for nodes (3.12). This result is in good agreement with the estimates (3.13) and (3.14) which, for $n = 21$ yield $\Lambda_n(\mathbf{x}) \lesssim 2.9008$ and $\Lambda_n(\mathbf{x}) \lesssim 2.9304$, respectively. ■

3.3.4 Barycentric interpolation formula

The interpolating polynomial $\Pi_n(x)$ introduced in Proposition 3.1 can be computed by the following *barycentric formula* ([BT04])

$$\boxed{\Pi_n(x) = \frac{\displaystyle\sum_{k=0}^{n} \frac{w_k}{x - x_k} y_k}{\displaystyle\sum_{k=0}^{n} \frac{w_k}{x - x_k}}} \tag{3.15}$$

where

$$w_k = \left(\prod_{\substack{j=0\\ j\neq k}}^{n} (x_k - x_j)\right)^{-1}, \qquad k = 0, \dots, n, \tag{3.16}$$

are called *barycentric weigths*.

In order to deduce (3.15) from (3.4), we rewrite the Lagrange characteristic polynomials (3.3) as

$$\varphi_k(x) = \prod_{\substack{j=0\\ j\neq k}}^{n} \frac{x - x_j}{x_k - x_j} = \underbrace{\left(\prod_{j=0}^{n} (x - x_j)\right)}_{\ell(x)} \frac{w_k}{x - x_k},$$

thus

$$\Pi_n(x) = \ell(x) \sum_{k=0}^{n} \frac{w_k}{x - x_k} y_k. \tag{3.17}$$

Noting that

$$\ell(x) \sum_{k=0}^{n} \frac{w_k}{x - x_k} = 1,$$

(this follows by (3.17) by taking $y_k = 1$ for $k = 0, \dots, n$ and noting that in this case $\Pi_n(x) \equiv 1$) it holds

$$\Pi_n(x) = \frac{\displaystyle \ell(x) \sum_{k=0}^{n} \frac{w_k}{x - x_k} y_k}{\displaystyle \ell(x) \sum_{k=0}^{n} \frac{w_k}{x - x_k}},$$

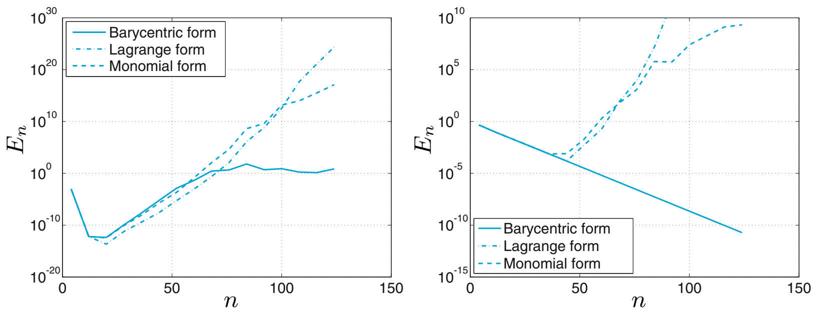


Figure 3.9. At left, interpolation errors for equispaced nodes in $[-1,1]$ and $f(x) = \sin(x)$. At right, interpolation errors for Chebyshev nodes in $[-5,5]$ and $f(x) = 1/(1+x^2)$

that is (3.15).

Consider a set of equispaced nodes and a function $f(x)$ such that $\max_{x\in[a,b]} |f(x) - \Pi_n f(x)| \to 0$ for $n \to \infty$ in exact arithmetics. Consider the interpolatory polynomial in Lagrange form (see (3.4)), in monomial form

$$\Pi_n f(x) = \sum_{k=0}^{n} c_{k+1} x^{n-k}, \tag{3.18}$$

and in barycentric form (see (3.15)). For all (three) cases, the interpolation error decreases until a certain value of n, then it starts increasing. It diverges for the Lagrange and monomial form, and keeps bounded (of the order of 1) for the barycentric form. See Figure 3.9, left, which corresponds to the interpolation of the function $f(x) = sin(x)$ on $[-1,1]$.

As a matter of fact, if the nodes are uniformely distributed, the weights of the barycentric formula are $w_k = (-1)^k \binom{n}{k}$.

This can potentially generate large oscillations of the interpolatory polynomial near the borders of the interval containing the interpolation nodes. However, this Runge's phenomenon (see Example 3.2) is not peculiar of the barycentric formula, as a matter of fact it is intrinsic to the Lagrange interpolation on equispaced nodes, as it is due to the bad conditioning when n gets large (because of the asymptotic behavior of the Lebesgue constant, see Sect. 3.3.2): small changes in the data can generate big changes in the interpolant.

On the contrary, when we use Chebyshev interpolatory nodes, the interpolation problem is well conditioned (see Sect. 3.3.3) and the weights of the interpolatory formula are bounded. While both the Lagrange and monomial forms are unstable with respect to the roundoff errors, the barycentric form is stable also for large values of n (see Fig. 3.9, on the right).

In conclusion, the main strength of the barycentric formula (3.15) is that it is stable with respect to the propagation of rounding errors, provided that two matters described below are attended to (see [BT04, Hig04]).

The first matter is concerned with underflow and overflow. When $n \to \infty$, the scale of the weights w_j (3.16) will grow or decay exponentially at the rate $(4/(b-a))^n$, where $[a, b]$ is the interval on which we seek the interpolating polynomial. Underflow and overflow can be avoided by multiplying each factor $(x_k - x_j)$ in (3.16) by $4/(b-a)$.

The second matter is about the evaluation of $\Pi_n(x)$ when $x = x_k$. A naïf implementation of (3.15) would yield `Nan` output, but it is sufficient to replace the computation (3.15) of $\Pi_n(x_k)$ with the given value y_k. When x is very close to x_k the barycentric formula turns out to be stable, as pointed out in [Hen79].

Program 3.1 implements the barycentric formula (3.15) by taking into account the tricks suggested above. The input variables `x`, `y`, and `x1` takes on the same meaning as in the call to the MATLAB function `polyfit`. The output variable `y1` contains the values of $\Pi_n(x)$ at nodes `x1`.

Program 3.1. barycentric: barycentric interpolation

```
function [y1]=barycentric(x,y,x1)
%BARYCENTRIC Computes the barycentric interpolating
% Y1=BARYCENTRIC(X,Y,X1) computes the value at the
% abscissae X1 of the polynomial interpolating data
% (X,Y), by using barycentric formula.
np=length(x);
a=min(x); b=max(x);
w=ones(np,1);
C=4/(b-a);
for j=1:np
    for k=1:j-1
        w(j)=w(j)*(x(j)-x(k))*C;
    end
    for k=j+1:np
        w(j)=w(j)*(x(j)-x(k))*C;
    end
end
w=1./w;
num=zeros(size(x1));den=num; exa=num;
for j=1:np
    xdiff=x1-x(j); wx=w(j)./xdiff;
    den=den+wx;  num=num+wx*y(j);
    exa(xdiff==0)=j;
end
y1=num./den;
for i=1:length(x1)
    if exa(i)>0, y1(i)=y(exa(i)); end
end
```

Example 3.6 We interpolate $f(x) = 1/(1+x^2)$ on the interval $[-5,5]$ at the $(n+1)$ Chebyshev-Gauss-Lobatto nodes (3.11) by calling Program 3.1, that implements barycentric formula (3.15), then a program that computes $\Pi_n f$ by using the Lagrange form (3.4), and finally the MATLAB command `polyfit` that computes the coefficients of $\Pi_n f$ with respect to the basis of monomials (see (3.18)).

According to the theory, the interpolation error $E_n = \max_{[-5,5]} |f(x) - \Pi_n f(x)|$ should converge exponentially to zero when $n \to \infty$, because of the nice properties of the Gaussian nodes.

In practice, as we can see from Figure 3.9, when $\Pi_n f(x)$ is computed by either the Lagrange formula (3.4) or the expansion (3.18), the error E_n decreases exponentially until $n \simeq 40$, while beyond it starts growing due to the propagation of rounding errors (for $n \geq 20$ a warning message is printed by MATLAB function `polyfit`, pointing out that "Polynomial is badly conditioned").

On the contrary, the error associated with the barycentric formula (3.15) keeps decreasing until machine epsilon. We have considered $n = 4:8:128$. ■

3.3.5 Trigonometric interpolation and FFT

We want to approximate a periodic function $f : [0, 2\pi] \to \mathbb{C}$, i.e. one satisfying $f(0) = f(2\pi)$, by a trigonometric polynomial $\tilde{f}$ which interpolates f at the equispaced $n+1$ nodes $x_j = 2\pi j/(n+1)$, $j = 0, \ldots, n$, i.e.

$$\tilde{f}(x_j) = f(x_j), \text{ for } j = 0, \ldots, n. \tag{3.19}$$

The *trigonometric interpolant* $\tilde{f}$ is obtained by a linear combination of sines and cosines.

Let us consider at first the case n even. Precisely we seek a function

$$\tilde{f}(x) = \frac{a_0}{2} + \sum_{k=1}^{M} \left[a_k \cos(kx) + b_k \sin(kx) \right], \tag{3.20}$$

with $M = n/2$, whose complex coefficients a_k, $k = 0, \ldots, M$ and b_k (for $k = 1, \ldots, M$) are unknown. By recalling the Euler formula $e^{ikx} = \cos(kx) + i \sin(kx)$, the trigonometric polynomial (3.20) can be written as

$$\tilde{f}(x) = \sum_{k=-M}^{M} c_k e^{ikx}, \tag{3.21}$$

where i is the imaginary unit and the coefficients c_k, for $k = 0, ..., M$, are related to the coefficient a_k and b_k through the formulas

$$a_k = c_k + c_{-k}, \qquad b_k = i(c_k - c_{-k}). \tag{3.22}$$

As a matter of fact, thanks to the parity properties of sine and cosine functions, it holds

$$\begin{aligned}\sum_{k=-M}^{M} c_k e^{ikx} &= \sum_{k=-M}^{M} c_k \left(\cos(kx) + i\sin(kx)\right)\\ &= c_0 + \sum_{k=1}^{M} \left[c_k(\cos(kx) + i\sin(kx)) + c_{-k}(\cos(kx) - i\sin(kx))\right]\\ &= c_0 + \sum_{k=1}^{M} \left[(c_k + c_{-k})\cos(kx) + i(c_k - c_{-k})\sin(kx))\right].\end{aligned}$$

When n is odd, the trigonometric polynomial $\tilde{f}$ can be defined as

$$\tilde{f}(x) = \sum_{k=-(M+1)}^{M+1} c_k e^{ikx}, \tag{3.23}$$

where $M = (n-1)/2$. Note that these are $n+2$ unknown coefficients in (3.23), while the interpolation conditions (3.19) are only $n+1$. A possible remedy consists of imposing $c_{-(M+1)} = c_{(M+1)}$, as done by MATLAB in the function `interpft`.
Even when n is odd we can write $\tilde{f}$ as a sum of sine and cosine functions, obtaining a formula similar to (3.20) in which the index k of the sum ranges now from 1 to $M+1$. Coefficients c_k in (3.23) are still related to coefficients a_k and b_k through the formulas (3.22), however now $k = 0, \ldots, M+1$. Due to the choice $c_{-(M+1)} = c_{(M+1)}$, we have $a_{(M+1)} = 2c_{(M+1)}$ and $b_{(M+1)} = 0$.

For the sake of generalization, we introduce a parameter μ that we set to 0, if n is even, and to 1, if n is odd. Then the interpolation polynomial can be written in a more general way as

$$\tilde{f}(x) = \sum_{k=-(M+\mu)}^{M+\mu} c_k e^{ikx} = \sum_{k=-M}^{M} c_k e^{ikx} + 2\mu c_{(M+1)} \cos((M+1)x),$$

where we recall that $M = (n-\mu)/2$ and c_{M+1} is meaningless for even n ($\mu = 0$).

Because of its analogy with Fourier series, $\tilde{f}$ is also named *discrete Fourier series* of f. By imposing interpolation conditions at nodes $x_j = jh$, with $h = 2\pi/(n+1)$, we find, for $j = 0, \ldots, n$,

$$\sum_{k=-M}^{M} c_k e^{ikjh} + 2\mu c_{(M+1)} \cos((M+1)jh) = f(x_j). \tag{3.24}$$

In order to compute the coefficients $\{c_k\}$, with $k = -M, \ldots, M+\mu$, we multiply equation (3.24) by $e^{-imx_j} = e^{-imjh}$ where m is an integer ranging between $-M$ and $M+\mu$, and then sum with respect to j

$$\begin{aligned}&\sum_{j=0}^{n}\sum_{k=-M}^{M} c_k e^{ikjh}e^{-imjh}\\&\qquad+2\mu c_{(M+1)}\sum_{j=0}^{n}\cos((M+1)jh)e^{-imjh}=\sum_{j=0}^{n}f(x_j)e^{-imjh}.\end{aligned}\tag{3.25}$$

Let us consider the identity

$$\sum_{j=0}^{n}e^{ijh(k-m)}=(n+1)\delta_{km},\qquad k,m=-M,\dots,M$$

which is obviously true if $k=m$. When $k\neq m$, it follows from the property

$$\sum_{j=0}^{n}e^{ijh(k-m)}=\frac{1-(e^{i(k-m)h})^{n+1}}{1-e^{i(k-m)h}},$$

and the remark that the numerator on the right hand side is null, since

$$\begin{aligned}1-e^{i(k-m)h(n+1)}&=1-e^{i(k-m)2\pi}\\&=1-\cos((k-m)2\pi)-i\sin((k-m)2\pi).\end{aligned}$$

By applying Euler formula and recalling the definitions of M ad h, it holds

$$\sum_{j=0}^{n}\cos((M+1)jh)e^{-imjh}=(n+1)\delta_{(M+1)m},$$

thus, from (3.25) we draw the following explicit expression for the coefficients of $\tilde f$

$$\boxed{\begin{array}{ll}c_k=\dfrac{1}{n+1}\displaystyle\sum_{j=0}^{n}f(x_j)e^{-ikjh}, & k=-M,\dots,M\\ c_{(M+1)}=c_{-(M+1)}=\dfrac{1}{2(n+1)}\displaystyle\sum_{j=0}^{n}(-1)^j f(x_j), & \text{only for odd } n\end{array}}\tag{3.26}$$

We deduce from (3.26) that, if f is a real valued function, then $c_{(M+1)}=c_{-(M+1)}$ are real and $c_{-k}=\overline{c_k}$ for $k=-M,\dots,M$ (this follows from $e^{ikjh}=\overline{e^{-ikjh}}$). In view of (3.22) we have $a_k,\ b_k\in\mathbb{R}$ (for $k=0,\dots,M+\mu$), thus $\tilde f$ is a real valued function, too.
The computation of all the coefficients $\{c_k\}$ can be accomplished with an order $n\log_2 n$ operations by using the *Fast Fourier Transform* (FFT), which is implemented in the MATLAB program `fft` (see Example 3.7). `fft`

Similar conclusions hold for the inverse transform through which we obtain the values $\{f(x_j)\}$ from the coefficients $\{c_k\}$. The inverse fast Fourier transform is implemented in the MATLAB program `ifft`. `ifft`

Example 3.7 Consider the function $f(x) = x(x-2\pi)e^{-x}$ for $x \in [0, 2\pi]$. To use the MATLAB program `fft` we first compute the values of f at the nodes $x_j = j\pi/5$ for $j = 0, \ldots, 9$ by the following instructions (recall that `.*` is the component-by-component vector product):

```
n=9; x=2*pi/(n+1)*[0:n]; y=x.*(x-2*pi).*exp(-x);
```

Now we compute by the FFT the vector of Fourier coefficients, with the following instructions:

```
Y=fft(y);
C=fftshift(Y)/(n+1)

C =
  Columns 1 through 2
   0.0870                  0.0926 - 0.0214i
  Columns 3 through 4
   0.1098 - 0.0601i   0.1268 - 0.1621i
  Columns 5 through 6
  -0.0467 - 0.4200i  -0.6520
  Columns 7 through 8
  -0.0467 + 0.4200i   0.1268 + 0.1621i
  Columns 9 through 10
   0.1098 + 0.0601i   0.0926 + 0.0214i
```

Elements of `Y` are related to coefficients c_k defined in (3.26) by the following relation: `Y`$= (n+1)[c_0, \ldots, c_M, c_{-(M+\mu)}, \ldots, c_{-1}]$. When n is odd, the coefficient $c_{(M+1)}$ (which coincides with $c_{-(M+1)}$) is neglected. The command `fftshift` sorts the elements of the input array, so that `C`$= [c_{-(M+\mu)}, \ldots, c_{-1}, c_0, \ldots, c_M]$. `fftshift`

Note that the program `ifft` achieves the maximum efficiency when n is a power of 2, even though it works for any value of n. ■

`interpft` The command `interpft` provides the trigonometric interpolant of a set of real data. It requires in input an integer m and a vector of values which represent the values taken by a function (periodic with period p) at the set of points $x_j = jp/(n+1)$, $j = 0, \ldots, n$. `interpft` returns the m real values of the trigonometric interpolant, obtained by the Fourier transform, at the nodes $t_i = ip/m$, $i = 0, \ldots, m-1$. For instance, let us reconsider the function of Example 3.7 in $[0, 2\pi]$ and take its values at 10 equispaced nodes $x_j = j\pi/5$, $j = 0, \ldots, 9$. The values of the trigonometric interpolant at, say, the 100 equispaced nodes $t_i = 2i\pi/100$, $i = 0, \ldots, 99$ can be obtained as follows (see Figure 3.10)

```
n=9; x=2*pi/(n+1)*[0:n]; y=x.*(x-2*pi).*exp(-x);
z=interpft(y,100);
```

In some cases the accuracy of trigonometric interpolation can dramatically downgrade, as shown in the following example.

Example 3.8 Let us approximate the function $f(x) = f_1(x) + f_2(x)$, with $f_1(x) = \sin(x)$ and $f_2(x) = \sin(5x)$, using nine equispaced nodes in the interval

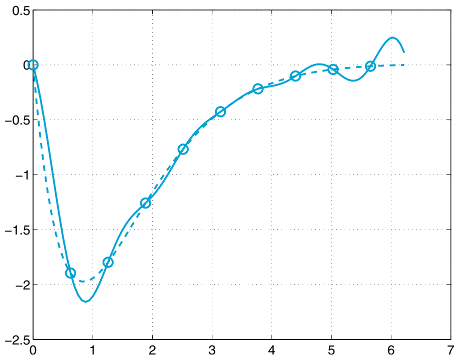


Figure 3.10. The function $f(x) = x(x - 2\pi)e^{-x}$ (*dashed line*) and the corresponding trigonometric interpolant (*solid line*) relative to 10 equispaced nodes

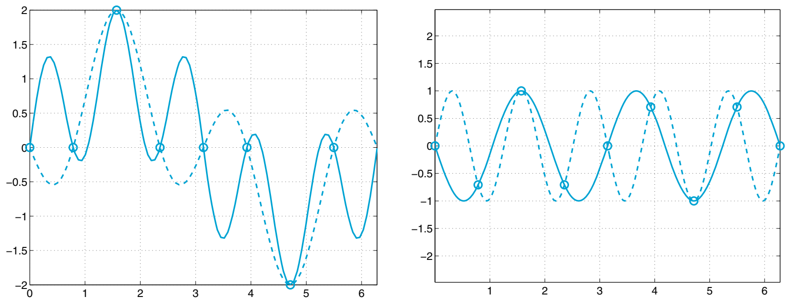


Figure 3.11. The effects of aliasing. At left, comparison between the function $f(x) = \sin(x) + \sin(5x)$ (*solid line*) and its trigonometric interpolant (3.20) with $M = 3$ (*dashed line*). At right, the functions $\sin(5x)$ (*dashed line*) and $-\sin(3x)$ (*solid line*) take the same values at the interpolation nodes. This circumstance explains the severe loss of accuracy shown at left

$[0, 2\pi]$. The result is shown in Figure 3.11, left. Note that in some intervals the trigonometric approximant shows even a phase inversion with respect to the function f. ■

This lack of accuracy can be explained as follows. At the nodes considered, the function f_2 is indistinguishable from $f_3(x) = -\sin(3x)$ which has a lower frequency (see Figure 3.11, right). The function that is actually approximated is therefore $F(x) = f_1(x) + f_3(x)$ and not $f(x)$ (in fact, the dashed line of Figure 3.11, left, does coincide with F).

This phenomenon is known as *aliasing* and may occur when the function to be approximated is the sum of several components having different frequencies. As soon as the number of nodes is not enough to resolve the highest frequencies, the latter may interfere with the low frequencies, giving rise to inaccurate interpolants. To get a better approximation for functions with higher frequencies, one has to increase the number of interpolation nodes.

A real life example of aliasing is provided by the apparent inversion of the sense of rotation of spoked wheels. Once a certain critical velocity is reached the human brain is no longer able to accurately sample the moving image and, consequently, produces distorted images.

We refer to Chapter 7 for the solution of nonlinear least squares problems, that is problems where the $\tilde{f}$ is a nonlinear function of the unknown coefficients a_j.

Let us summarize

1. Approximating a set of data or a function f in $[a, b]$ consists of finding a suitable function $\tilde{f}$ that represents them with enough accuracy;
2. the interpolation process consists of determining a function $\tilde{f}$ such that $\tilde{f}(x_i) = y_i$, where the $\{x_i\}$ are given nodes and $\{y_i\}$ are either the values $\{f(x_i)\}$ or a set of prescribed values;
3. if the $n+1$ nodes $\{x_i\}$ are distinct, there exists a unique polynomial of degree less than or equal to n interpolating a set of prescribed values $\{y_i\}$ at the nodes $\{x_i\}$;
4. for an equispaced distribution of nodes in $[a, b]$ the interpolation error at any point of $[a, b]$ does not necessarily tend to 0 as n tends to infinity. However, there exist special distributions of nodes, for instance the Chebyshev nodes, for which this convergence property holds true for all continuously differentiable functions;
5. trigonometric interpolation is well suited to approximate periodic functions, and is based on choosing $\tilde{f}$ as a linear combination of sine and cosine functions. The FFT is a very efficient algorithm which allows the computation of the Fourier coefficients of a trigonometric interpolant from its node values and admits an equally fast inverse, the IFFT.

3.4 Piecewise linear interpolation

The interpolant at Chebyshev nodes provides an accurate approximation of any smooth function f whose expression is known. In the case when f is nonsmooth or when f is only known through its values at a set of given points (which do not coincide with the Chebyshev nodes), one can resort to a different interpolation method which is called linear composite interpolation.

More precisely, given a distribution (not necessarily uniform) of nodes $x_0 < x_1 < \ldots < x_n$, we denote by I_i the interval $[x_i, x_{i+1}]$. We approximate f by a continuous function which, on each interval, is given by

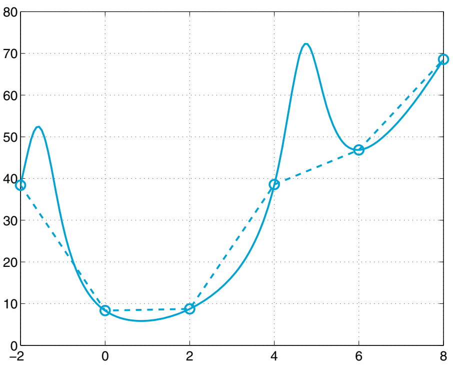


Figure 3.12. The function $f(x) = x^2 + 10/(\sin(x) + 1.2)$ (*solid line*) and its piecewise linear interpolation polynomial $\Pi_1^H f$ (*dashed line*)

the segment joining the two points $(x_i, f(x_i))$ and $(x_{i+1}, f(x_{i+1}))$ (see Figure 3.12). This function, denoted by $\Pi_1^H f$, is called *piecewise linear interpolation polynomial* of f and its expression is:

$$\Pi_1^H f(x) = f(x_i) + \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i}(x - x_i) \qquad \text{for } x \in I_i.$$

The upper-index H denotes the maximum length of the intervals I_i.

The following result can be inferred from (3.7) setting $n = 1$ and $h = H$:

Proposition 3.3 *If $f \in C^2(I)$, where $I = [x_0, x_n]$, then*

$$\max_{x\in I}|f(x) - \Pi_1^H f(x)| \leq \frac{H^2}{8}\max_{x\in I}|f''(x)|.$$

Consequently, for all x in the interpolation interval, $\Pi_1^H f(x)$ tends to $f(x)$ when $H \to 0$, provided that f is sufficiently smooth.

Through the instruction `s1=interp1(x,y,z)` one can compute the values at arbitrary points, which are stored in the vector `z`, of the piecewise linear polynomial that interpolates the values `y(i)` at the nodes `x(i)`, for `i = 1,...,n+1`. Note that `z` can have arbitrary dimension. If the nodes are in increasing order (i.e. `x(i+1) > x(i)`, for `i=1,...,n`) then we can use the quicker version `interp1q` (`q` stands for quickly). Notice that `interp1q` is quicker than `interp1` on non-uniformly spaced data because it does not make any input checking, nevertheless, we note that all input variables of `interp1q` must be column vectors.

`interp1`

`interp1q`

It is worth mentioning that the command `fplot`, which is used to display the graph of a function f on a given interval $[a, b]$, does indeed replace the function by its piecewise linear interpolant. The set of interpolating nodes is generated automatically from the function, following the criterion of clustering these nodes around points where f shows strong variations. A procedure of this type is called *adaptive*.

3.5 Approximation by spline functions

As done for piecewise linear interpolation, piecewise polynomial interpolation of degree $n \geq 2$ can be defined as well. For instance, the piecewise quadratic interpolation $\Pi_2^H f$ is a continuous function that on each interval I_i replaces f by its quadratic interpolation polynomial at the endpoints of I_i and at its midpoint. If $f \in C^3(I)$, the error $f - \Pi_2^H f$ in the maximum norm decays as H^3 if H tends to zero.

The main drawback of this piecewise interpolation is that $\Pi_k^H f$ with $k \geq 1$, is nothing more than a global continuous function. As a matter of fact, in several applications, e.g. in computer graphics, it is desirable to get approximation by smooth functions which have at least a continuous derivative.

With this aim, we can construct a function s_3 with the following properties:

1. on each interval $I_i = [x_i, x_{i+1}]$, for $i = 0, \ldots, n-1$, s_3 is a polynomial of degree 3 which interpolates the pairs of values $(x_j, f(x_j))$ for $j = i, i+1$ (s_3 is therefore a globally continuous function);
2. s_3 has continuous first and second derivatives in the nodes x_i, $i = 1, \ldots, n-1$.

For its complete determination, we need four conditions on each interval, therefore a total of $4n$ equations, which we can provide as follows:

- $n+1$ conditions arise from the interpolation requirement at the nodes x_i, $i = 0, \ldots, n$;
- $n-1$ further equations follow from the requirement of continuity of the polynomial at the internal nodes $x_1, \ldots, x_{n-1}$;
- $2(n-1)$ new equations are obtained by requiring that both first and second derivatives be continuous at the internal nodes.

We still lack two further equations, which we can e.g. choose as

$$s_3''(x_0) = 0,\ s_3''(x_n) = 0. \tag{3.27}$$

The function s_3 which we obtain in this way, is called a *natural interpolating cubic spline*.

By suitably choosing the unknowns (see [QSS07, Section 8.7]) to represent s_3 we arrive at a $(n+1) \times (n+1)$ system with a tridiagonal matrix whose solution can be accomplished by a number of operations proportional to n (see Section 5.6) whose solutions are the values $s''(x_i)$ for $i = 0, \dots, n$.

Using Program 3.2, this solution can be obtained with a number of operations equal to the dimension of the system itself (see Section 5.6). The input parameters are the vectors `x` and `y` of the nodes and the data to interpolate, plus the vector `zi` of the abscissae where we want the spline s_3 to be evaluated.

Other conditions can be chosen in place of (3.27) in order to close the system of equations; for instance we could prescribe the value of the first derivative of s_3 at both endpoints x_0 and x_n.

Unless otherwise specified, Program 3.2 computes the natural interpolation cubic spline. The optional parameters `type` and `der` (a vector with two components) serve the purpose of selecting other types of splines. With `type=0` Program 3.2 computes the interpolating cubic spline whose first derivative is given by `der(1)` at x_0 and `der(2)` at x_n. With `type=1` we obtain the interpolating cubic spline whose values of the second derivative at the endpoints is given by `der(1)` at x_0 and `der(2)` at x_n.

Program 3.2. cubicspline: interpolating cubic spline

```
function s=cubicspline(x,y,zi,type,der)
%CUBICSPLINE Computes a cubic spline
% S=CUBICSPLINE(X,Y,ZI) computes the value at the
% abscissae ZI of the natural interpolating cubic
% spline that interpolates the values Y at the nodes X.
% S=CUBICSPLINE(X,Y,ZI,TYPE,DER) if TYPE=0 computes the
% values at the abscissae ZI of the cubic spline
% interpolating the values Y with first derivative at
% the endpoints equal to the values DER(1) and DER(2).
% If TYPE=1 the values DER(1) and DER(2) are those of
% the second derivative at the endpoints.
[n,m]=size(x);
if n == 1
    x = x';    y = y';    n = m;
end
if nargin == 3
    der0 = 0; dern = 0; type = 1;
else
    der0 = der(1); dern = der(2);
end
h = x(2:end)-x(1:end-1);
e = 2*[h(1); h(1:end-1)+h(2:end); h(end)];
A = spdiags([[h; 0] e [0; h]],-1:1,n,n);
d = (y(2:end)-y(1:end-1))./h;
rhs = 3*(d(2:end)-d(1:end-1));
if type == 0
    A(1,1) = 2*h(1);   A(1,2) = h(1);
    A(n,n) = 2*h(end); A(end,end-1) = h(end);
    rhs = [3*(d(1)-der0); rhs; 3*(dern-d(end))];
else
    A(1,:) = 0; A(1,1) = 1;
    A(n,:) = 0; A(n,n) = 1;
```

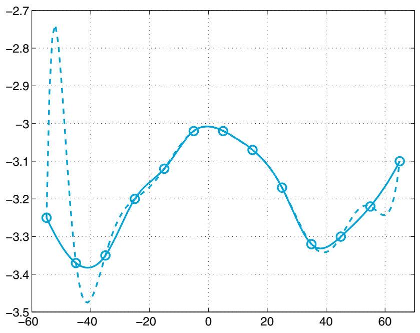


Figure 3.13. Comparison between the interpolating cubic spline (*solid line*) and the Lagrange interpolant (*dashed line*) for the case considered in Example 3.9

```
    rhs = [der0; rhs; dern];
end
S = zeros(n,4);
S(:,3) = A\rhs;
for m = 1:n-1
    S(m,4) = (S(m+1,3)-S(m,3))/3/h(m);
    S(m,2) = d(m) - h(m)/3*(S(m + 1,3)+2*S(m,3));
    S(m,1) = y(m);
end
S = S(1:n-1, 4:-1:1);
pp = mkpp(x,S);  s = ppval(pp,zi);
return
```

spline

The MATLAB command `spline` (see also the toolbox `splines`) enforces the third derivative of s_3 to be continuous at x_1 and x_{n-1}. To this condition is given the curious name of *not-a-knot condition*. The input parameters are the vectors `x` and `y` and the vector `zi` (same meaning as before). The commands `mkpp` and `ppval` that are used in Program 3.2 are useful to build up and evaluate a composite polynomial.

mkpp
ppval

Example 3.9 Let us reconsider the data of Table 3.1 corresponding to the column $K = 0.67$ and compute the associated interpolating cubic spline s_3. The different values of the latitude provide the nodes x_i, $i = 0, \ldots, 12$. If we are interested in computing the values $s_3(z_i)$, where $z_i = -55 + i$, $i = 0, \ldots, 120$, we can proceed as follows:

```
x = [-55:10:65];
y = [-3.25 -3.37 -3.35 -3.2 -3.12 -3.02 -3.02 ...
     -3.07 -3.17 -3.32 -3.3 -3.22 -3.1];
zi = [-55:1:65];
s = spline(x,y,zi);
```

The graph of s_3, which is reported in Figure 3.13, looks more plausible than that of the Lagrange interpolant at the same nodes. ■

Example 3.10 (Robotics) To find the trajectory in the xy plane of the robot satisfying the given constraints (see Problem 3.4), we split the time interval $[0,5]$ in the two subintervals $[0,2]$ and $[2,5]$. Then in each subinterval we look for two splines, $x = x(t)$ and $y = y(t)$, that interpolate the given values and have null derivative at the endpoints. Using Program 3.2 we obtain the desired result by the following instructions:

```
x1 = [0 1 4]; y1 = [0 2 4];
t1 = [0 1 2]; ti1 = [0:0.01:2];
x2 = [0 3 4]; y2 = [0 1 4];
t2 = [0 2 3]; ti2 = [0:0.01:3]; d=[0,0];
six1 = cubicspline(t1,x1,ti1,0,d);
siy1 = cubicspline(t1,y1,ti1,0,d);
six2 = cubicspline(t2,x2,ti2,0,d);
siy2 = cubicspline(t2,y2,ti2,0,d);
```

The trajectory obtained is drawn in Figure 3.14. ■

The error that we obtain in approximating a function f (continuously differentiable up to its fourth derivative) by the natural interpolating cubic spline s_3 satisfies the following inequalities ([dB01]):

$$\max_{x\in I}|f^{(r)}(x) - s_3^{(r)}(x)| \leq C_r H^{4-r} \max_{x\in I}|f^{(4)}(x)|, \quad r = 0,1,2,$$

and

$$\max_{x\in I\setminus\{x_0,\ldots,x_n\}}|f^{(3)}(x) - s_3^{(3)}(x)| \leq C_3 H \max_{x\in I}|f^{(4)}(x)|,$$

where $I = [x_0, x_n]$ and $H = \max_{i=0,\ldots,n-1}(x_{i+1} - x_i)$, while C_r (for $r = 0,\ldots,3$) is a suitable constant depending on r, but independent of H. It is then clear that not only f, but also its first, second and third derivatives are well approximated by s_3 when H tends to 0.

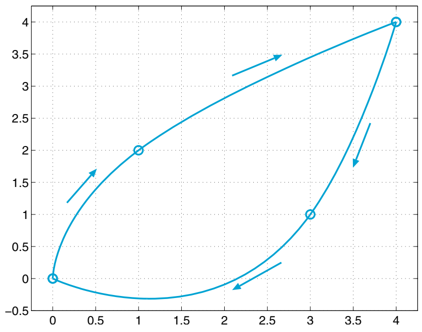


Figure 3.14. The trajectory in the xy plane of the robot described in Problem 3.4. Circles represent the position of the control points through which the robot should pass during its motion

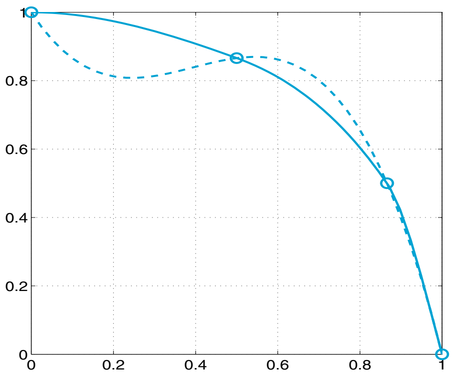


Figure 3.15. Approximation of the first quarter of the circumference of the unitary circle using only 4 nodes. The dashed line is the cubic spline, while the solid line is the piecewise cubic Hermite interpolant

Remark 3.1 In general cubic splines do not preserve monotonicity between neighbouring nodes. For instance, by approximating the unitary circumference in the first quarter using the points $(x_k = \sin(k\pi/6), y_k = \cos(k\pi/6))$, for $k = 0, \dots, 3$, we would obtain an oscillatory spline (see Figure 3.15). In these cases, other approximation techniques can be better suited. For instance, the MATLAB command `pchip` provides the Hermite piecewise cubic interpolant ([Atk89]) which is locally monotone and interpolates the function as well as its first derivative at the nodes $\{x_i, i = 1, \dots, n-1\}$ (see Figure 3.15). The Hermite interpolant can be obtained by using the following instructions:

pchip

```
t = linspace(0,pi/2,4);
x = sin(t); y = cos(t);
xx = linspace(0,1,40);
plot(x,y,'o',xx,[pchip(x,y,xx);spline(x,y,xx)])
```

■

See the Exercises 3.5-3.8.

3.6 The least-squares method

As already noticed, a Lagrange interpolation does not guarantee a better approximation of a given function when the polynomial degree gets large. This problem can be overcome by composite interpolation (such as piecewise linear polynomials or splines). However, neither are suitable to extrapolate information from the available data, that is, to generate new values at points lying outside the interval where interpolation nodes are given.

Example 3.11 (Finance) On the basis of the data reported in Figure 3.1, we would like to predict whether the stock price will increase or diminish in the coming days. The Lagrange polynomial interpolation is impractical, as it would require a (tremendously oscillatory) polynomial of degree 719 which will provide a completely erroneous prediction. On the other hand, piecewise linear interpolation, whose graph is reported in Figure 3.1, provides extrapolated results by exploiting only the values of the last two days, thus completely neglecting the previous history. To get a better result we should avoid the interpolation requirement, by invoking least-squares approximation as indicated below. ■

Assume that the data $\{(x_i, y_i), i = 0, \ldots, n\}$ are available, where now y_i could represent the values $f(x_i)$ attained by a given function f at the nodes x_i. For a given integer $m \geq 1$ (usually, $m \ll n$) we look for a polynomial $\tilde{f} \in \mathbb{P}_m$ which satisfies the inequality

$$\boxed{\sum_{i=0}^{n}[y_i - \tilde{f}(x_i)]^2 \leq \sum_{i=0}^{n}[y_i - p_m(x_i)]^2} \tag{3.28}$$

for every polynomial $p_m \in \mathbb{P}_m$. Should it exist, $\tilde{f}$ will be called the *least-squares approximation* in $\mathbb{P}_m$ of the set of data $\{(x_i, y_i), i = 0, \ldots, n\}$. Unless $m \geq n$, in general it will not be possible to guarantee that $\tilde{f}(x_i) = y_i$ for all $i = 0, \ldots, n$.

Setting

$$\tilde{f}(x) = a_0 + a_1 x + \ldots + a_m x^m, \tag{3.29}$$

where the coefficients $a_0, \ldots, a_m$ are unknown, the problem (3.28) can be restated as follows: find $a_0, a_1, \ldots, a_m$ such that

$$\Phi(a_0, a_1, \ldots, a_m) = \min_{\{b_i,\ i=0,\ldots,m\}} \Phi(b_0, b_1, \ldots, b_m)$$

where

$$\Phi(b_0, b_1, \ldots, b_m) = \sum_{i=0}^{n} \left[y_i - (b_0 + b_1 x_i + \ldots + b_m x_i^m)\right]^2.$$

We solve this problem in the special case when $m = 1$. Since

$$\Phi(b_0, b_1) = \sum_{i=0}^{n} \left[y_i^2 + b_0^2 + b_1^2 x_i^2 + 2b_0 b_1 x_i - 2b_0 y_i - 2b_1 x_i y_i\right],$$

the graph of Φ is a convex paraboloid. The point (a_0, a_1) at which Φ attains its minimum satisfies the conditions

$$\frac{\partial \Phi}{\partial b_0}(a_0, a_1) = 0, \qquad \frac{\partial \Phi}{\partial b_1}(a_0, a_1) = 0,$$

where the symbol $\partial\Phi/\partial b_j$ denotes the partial derivative (that is, the rate of variation) of Φ with respect to b_j, after having frozen the remaining variable (see the definition (9.3)).
By explicitly computing the two partial derivatives we obtain

$$\sum_{i=0}^{n}[a_0 + a_1 x_i - y_i] = 0, \qquad \sum_{i=0}^{n}[a_0 x_i + a_1 x_i^2 - x_i y_i] = 0,$$

which is a system of two equations for the two unknowns a_0 and a_1:

$$\begin{aligned} a_0(n+1) + a_1\sum_{i=0}^{n} x_i &= \sum_{i=0}^{n} y_i, \\ a_0\sum_{i=0}^{n} x_i + a_1\sum_{i=0}^{n} x_i^2 &= \sum_{i=0}^{n} y_i x_i. \end{aligned} \tag{3.30}$$

Setting $D = (n+1)\sum_{i=0}^{n} x_i^2 - (\sum_{i=0}^{n} x_i)^2$, the solution reads:

$$\begin{aligned} a_0 &= \frac{1}{D}\left[\sum_{i=0}^{n} y_i \sum_{j=0}^{n} x_j^2 - \sum_{j=0}^{n} x_j \sum_{i=0}^{n} x_i y_i\right], \\ a_1 &= \frac{1}{D}\left[(n+1)\sum_{i=0}^{n} x_i y_i - \sum_{j=0}^{n} x_j \sum_{i=0}^{n} y_i\right] \end{aligned} \tag{3.31}$$

The corresponding polynomial $\tilde{f}(x) = a_0 + a_1 x$ is known as the *least-squares straight line*, or *regression line*.

The previous approach can be generalized in several ways. The first generalization is to the case of an arbitrary m. The associated $(m+1)\times(m+1)$ linear system, which is symmetric, will have the form:

$$\begin{array}{llll} a_0(n+1) & +a_1\sum_{i=0}^{n} x_i & +\ldots + a_m\sum_{i=0}^{n} x_i^m & = \sum_{i=0}^{n} y_i, \\ a_0\sum_{i=0}^{n} x_i & +a_1\sum_{i=0}^{n} x_i^2 & +\ldots + a_m\sum_{i=0}^{n} x_i^{m+1} & = \sum_{i=0}^{n} x_i y_i, \\ \vdots & \vdots & \vdots & \vdots \\ a_0\sum_{i=0}^{n} x_i^m & +a_1\sum_{i=0}^{n} x_i^{m+1} & +\ldots + a_m\sum_{i=0}^{n} x_i^{2m} & = \sum_{i=0}^{n} x_i^m y_i. \end{array}$$

When $m = n$, the least-squares polynomial $\tilde{f}$ must coincide with the Lagrange interpolating polynomial $\Pi_n f$ (see Exercise 3.9).

The MATLAB command `c=polyfit(x,y,m)` computes by default the coefficients of the polynomial of degree `m` which approximates `n+1` pairs of data `(x(i),y(i))` in the least-squares sense. As already noticed in Section 3.3.1, when `m` is equal to `n` it returns the interpolating polynomial.

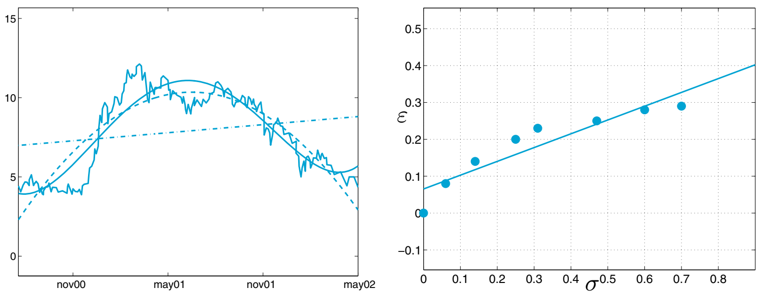


Figure 3.16. At left: least-squares approximation of the data of Problem 3.2 with polynomials of degree 1 (*dashed-dotted line*), degree 2 (*dashed line*) and degree 4 (*thick solid line*). The exact data are represented by the *thin solid line*. At right: linear least-squares approximation of the data of Problem 3.3

Example 3.12 (Finance) In Figure 3.16, left, we draw the graphs of the least-squares polynomials of degree 1, 2 and 4 that approximate in the least-squares sense the data of Figure 3.1. The polynomial of degree 4 reproduces quite reasonably the behavior of the stock price in the considered time interval and suggests that in the near future the quotation will increase. ■

Example 3.13 (Biomechanics) Using the least-squares method we can answer the question in Problem 3.3 and discover that the line which better approximates the given data has equation $\epsilon(\sigma) = 0.3471\sigma + 0.0654$ (see Figure 3.16, right); when $\sigma = 0.9$ it provides the estimate $\epsilon = 0.2915$ for the deformation. ■

A further generalization of the least-squares approximation consists of using in (3.28) $\tilde{f}$ and p_m that are no-longer polynomials but functions of a space V_m obtained by linearly combining $m+1$ independent functions $\{\psi_j, j = 0, \dots, m\}$. Special instances are provided, e.g., by the trigonometric functions $\psi_j(x) = \cos(\gamma j x)$ (for a given parameter $\gamma \neq 0$), by the exponential functions $\psi_j(x) = e^{\delta j x}$ (for some $\delta > 0$), or by a suitable set of spline functions.

The choice of the functions $\{\psi_j\}$ is actually dictated by the conjectured behavior of the law underlying the given data distribution. For instance, in Figure 3.17 we draw the graph of the least-squares approximation of the data of the Example 3.1 computed using the trigonometric functions $\psi_j(x) = \cos(\gamma j x)$, $j = 0, \dots, 4$, with $\gamma = \pi/60$.

The reader can verify that the unknown coefficients of

$$\tilde{f}(x) = \sum_{j=0}^{m} a_j \psi_j(x),$$

can be obtained by solving the following system (of *normal equations*)

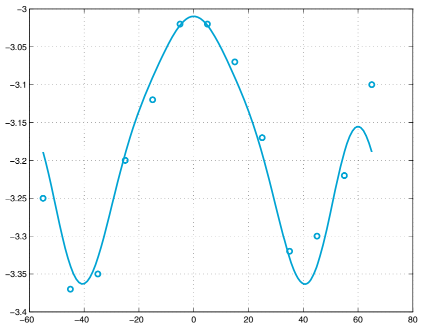


Figure 3.17. The least-squares approximation of the data of the Problem 3.1 using a cosine basis. The exact data are represented by the small circles

$$\boxed{\mathrm{B}^T\mathrm{B}\mathbf{a} = \mathrm{B}^T\mathbf{y}} \tag{3.32}$$

where B is the rectangular matrix $(n+1)\times(m+1)$ of entries $b_{ij} = \psi_j(x_i)$, $\mathbf{a}$ is the vector of the unknown coefficients, while $\mathbf{y}$ is the vector of the data. The linear system (3.32) can be efficiently solved by the QR factorization or, alternativeley, by a Singular-Value Decomposition of matrix B (see Section 5.7).

Let us summarize

1. The composite piecewise linear interpolant of a function f is a piecewise continuous linear function $\tilde{f}$, which interpolates f at a given set of nodes $\{x_i\}$. With this approximation we avoid Runge's type phenomena when the number of nodes increases. It is also called piecewise linear *finite element interpolant* (see Chapter 9);
2. interpolation by cubic splines allows the approximation of f by a piecewise cubic function $\tilde{f}$ which is continuous together with its first and second derivatives;
3. in least-squares approximation we look for an approximant $\tilde{f}$ which is a polynomial of degree m (typically, $m \ll n$) that minimizes the mean-square error $\sum_{i=0}^{n}[y_i - \tilde{f}(x_i)]^2$. The same minimization criterium can be applied for a class of functions that are not polynomials.

See the Exercises 3.9-3.14.

3.7 What we haven't told you

For a more general introduction to the theory of interpolation and approximation the reader is referred to, e.g., [Dav63], [Mei67] and [Gau97].

Polynomial interpolation can also be used to approximate data and functions in several dimensions. In particular, composite interpolation, based on piecewise linear or spline functions, is well suited when the region Ω at hand is partitioned into polygons in 2D (triangles or quadrilaterals) and polyhedra in 3D (tetrahedra or prisms).

A special situation occurs when Ω is a rectangle or a parallelepiped in which case the MATLAB commands interp2, and interp3, respectively, can be used. In both cases it is assumed that we want to represent on a regular, fine lattice (or grid) a function whose values are available on a regular, coarser lattice.

interp2
interp3

Consider for instance the values of $f(x, y) = \sin(2\pi x)\cos(2\pi y)$ on a (coarse) 6×6 lattice of equispaced nodes on the square $[0, 1]^2$; these values can be obtained using the commands:

```
[x,y]=meshgrid(0:0.2:1,0:0.2:1);
z=sin(2*pi*x).*cos(2*pi*y);
```

By the command interp2 a cubic spline is first computed on this coarse grid, then evaluated at the nodal points of a finer grid of 21×21 equispaced nodes:

```
xi = [0:0.05:1]; yi=[0:0.05:1];
[xf,yf]=meshgrid(xi,yi);
pi3=interp2(x,y,z,xf,yf);
```

The command meshgrid transforms the set of the couples (xi(k), yi(j)) into two matrices xf and yf that can be used to evaluate functions of two variables and to plot three dimensional surfaces. The rows of xf are copies of the vector xi, the columns of yf are copies of yi. Alternatively to the above procedure we can use the command griddata, available also for three-dimensional data (griddata3) and for the approximation of n-dimensional surfaces (griddatan).

meshgrid

griddata

The commands described below are for MATLAB only.
When Ω is a two-dimensional domain of (almost) arbitrary shape, it can be partitioned into triangles using the graphical interface pdetool.

pdetool

For a general presentation of spline functions see, e.g., [Die93] and [PBP02]. The MATLAB toolbox splines allows one to explore several applications of spline functions. In particular, the spdemos command gives the user the possibility to investigate the properties of the most important type of spline functions. Rational splines, i.e. functions which are the ratio of two splines functions, are accessible through the commands rpmak and rsmak. Special instances are the so-called NURBS splines, which are commonly used in CAGD (*Computer Assisted Geometric Design*).

spdemos

rpmak
rsmak

In the same context of Fourier approximation, we mention the approximation based on *wavelets*. This type of approximation is largely used for image reconstruction and compression and in signal analysis (for an introduction, see [DL92], [Urb02]). A rich family of wavelets (and their applications) can be found in the MATLAB toolbox wavelet.

wavelet

Octave 3.3 The Octave-Forge Package `msh` provides an interface for importing into the Octave workspace triangular or tetrahedral meshes generated with the graphical interface of GMSH (`http://geuz.org/gmsh/`). There is a `splines` package in Octave-Forge but it has limited functionality and does not provide the `spdemos` command.
The Octave-Forge package `nurbs` provides a set of functions for creating and managing NURBS surfaces and volumes. ■

3.8 Exercises

Exercise 3.1 Prove inequality (3.6).

Exercise 3.2 Provide an upper bound of the Lagrange interpolation error for the following functions:

$$\begin{array}{ll} f_1(x) = \cosh(x),\ f_2(x) = \sinh(x), & x_k = -1 + 0.5k,\ k = 0, \dots, 4, \\ f_3(x) = \cos(x) + \sin(x), & x_k = -\pi/2 + \pi k/4,\ k = 0, \dots, 4. \end{array}$$

Exercise 3.3 The following data are related to the life expectation of citizens of two European regions:

Year	1975	1980	1985	1990
Western Europe	72.8	74.2	75.2	76.4
Eastern Europe	70.2	70.2	70.3	71.2

Use the interpolating polynomial of degree 3 to estimate the life expectation in 1977, 1983 and 1988.

Exercise 3.4 The price (in euros) of a magazine has changed as follows:

Nov.87	Dec.88	Nov.90	Jan.93	Jan.95	Jan.96	Nov.96	Nov.00
4.5	5.0	6.0	6.5	7.0	7.5	8.0	8.0

Estimate the price in November 2002 by extrapolating these data.

Exercise 3.5 Repeat the computations carried out in Exercise 3.3, using now the cubic interpolating spline computed by the function `spline`. Then compare the results obtained with those obtained by solving Exercise 3.3.

Exercise 3.6 In the table below we report the values of the sea water density ρ (in Kg/m^3) corresponding to different values of the temperature T (in degrees Celsius):

T	4^o	8^o	12^o	16^o	20^o
ρ	1000.7794	1000.6427	1000.2805	999.7165	998.9700

Compute the cubic spline s_3 on the interval $4 \leq T \leq 20$, divided into 4 equal subintervals. Then compare the results provided by the spline interpolant with the following ones (which correspond to further values of T):

T	6^o	10^o	14^o	18^o
ρ	1000.74088	1000.4882	1000.0224	999.3650

Exercise 3.7 The Italian production of citrus fruit has changed as follows:

Year	1965	1970	1980	1985	1990	1991
production ($\times 10^5$ Kg)	17769	24001	25961	34336	29036	33417

Use interpolating cubic splines of different kinds to estimate the production in 1962, 1977 and 1992. Compare these results with the real values: 12380, 27403 and 32059 ($\times 10^5$ Kg), respectively. Compare the results with those that would be obtained using the Lagrange interpolating polynomial.

Exercise 3.8 Evaluate the function $f(x) = \sin(2\pi x)$ at 21 equispaced nodes in the interval $[-1, 1]$. Compute the Lagrange interpolating polynomial and the cubic interpolating spline. Compare the graphs of these two functions with that of f on the given interval. Repeat the same calculation using the following perturbed set of data: $f(x_i) = (-1)^{i+1}10^{-4}$ $(i = 0, \ldots, n)$, and observe that the Lagrange interpolating polynomial is more sensitive to small perturbations than the cubic spline.

Exercise 3.9 Verify that if $m = n$ the least-squares polynomial of a function f at the nodes $x_0, \ldots, x_n$ coincides with the interpolating polynomial $\Pi_n f$ at the same nodes.

Exercise 3.10 Compute the least-squares polynomial of degree 4 that approximates the values of K reported in the different columns of Table 3.1.

Exercise 3.11 Repeat the computations carried out in Exercise 3.7 using now a least-squares approximation of degree 3.

Exercise 3.12 Express the coefficients of system (3.30) in terms of the *average* $M = \frac{1}{(n+1)} \sum_{i=0}^{n} x_i$ and the *variance* $v = \frac{1}{(n+1)} \sum_{i=0}^{n} (x_i - M)^2$ of the set of data $\{x_i, i = 0, \ldots, n\}$.

Exercise 3.13 Verify that the regression line passes through the point whose abscissa is the average of $\{x_i\}$ and ordinate is the average of y_i.

Exercise 3.14 The following values

Flow rate	0	35	0.125	5	0	5	1	0.5	0.125	0

represent the measured values of the blood flow-rate in a cross-section of the carotid artery during a heart beat. The frequency of acquisition of the data is constant and is equal to $10/T$, where $T = 1$ s is the beat period. Represent these data by a continuous function of period equal to T.

4

Numerical differentiation and integration

In this chapter we propose methods for the numerical approximation of derivatives and integrals of functions. Concerning integration, quite often for a generic function it is not possible to find a primitive in an explicit form. Even when a primitive is known, its use might not be easy. This is, e.g., the case of the function $f(x) = \cos(4x)\cos(3\sin(x))$, for which we have

$$\int_0^\pi f(x)dx = \pi\left(\frac{3}{2}\right)^4 \sum_{k=0}^{\infty} \frac{(-9/4)^k}{k!(k+4)!};$$

the task of computing an integral is transformed into the equally troublesome one of summing a series. In other circumstances the function that we want to integrate or differentiate could only be known on a set of nodes (for instance, when the latter represent the results of an experimental measurement), exactly as happens in the case of function approximation, which was discussed in Chapter 3.

In all these situations it is necessary to consider numerical methods in order to obtain an approximate value of the quantity of interest, independently of how difficult is the function to integrate or differentiate.

4.1 Some representative problems

Problem 4.1 (Hydraulics) The height $q(t)$ reached at time t by a fluid in a straight cylinder of radius $R = 1$ m with a circular hole of radius $r = 0.1$ m on the bottom, has been measured every 5 seconds yielding the following values

t	0	5	10	15	20
$q(t)$	0.6350	0.5336	0.4410	0.3572	0.2822

We want to compute an approximation of the emptying velocity $q'(t)$ of the cylinder, then compare it with the one predicted by Torricelli's law: $q'(t) = -\gamma(r/R)^2\sqrt{2gq(t)}$, where g is the modulus of gravity acceleration and $\gamma = 0.6$ is a correction factor. For the solution of this problem, see Example 4.1. ■

Problem 4.2 (Optics) In order to plan a room for infrared beams we are interested in calculating the energy emitted by a black body (that is, an object capable of irradiating in all the spectrum to the ambient temperature) in the (infrared) spectrum comprised between 3μm and 14μm wavelength. The solution of this problem is obtained by computing the integral

$$E(T) = 2.39 \cdot 10^{-11} \int\limits_{3\cdot 10^{-4}}^{14\cdot 10^{-4}} \frac{dx}{x^5(e^{1.432/(Tx)} - 1)}, \tag{4.1}$$

which is the Planck equation for the energy $E(T)$, where x is the wavelength (in cm) and T the temperature (in Kelvin) of the black body. For its computation see Exercise 4.17. ■

Problem 4.3 (Electromagnetism) Consider an electric wire sphere of arbitrary radius r and conductivity σ. We want to compute the density distribution of the current $\mathbf{j}$ as a function of r and t (the time), knowing the initial distribution of the charge density $\rho(r)$. The problem can be solved using the relations between the current density, the electric field and the charge density and observing that, for the symmetry of the problem, $\mathbf{j}(r,t) = j(r,t)\mathbf{r}/|\mathbf{r}|$, where $j = |\mathbf{j}|$. We obtain

$$j(r,t) = \gamma(r)e^{-\sigma t/\varepsilon_0},\ \gamma(r) = \frac{\sigma}{\varepsilon_0 r^2}\int\limits_0^r \rho(\xi)\xi^2\, d\xi, \tag{4.2}$$

where $\varepsilon_0 = 8.859 \cdot 10^{-12}$ farad/m is the dielectric constant of the void. For the computation of this integral, see Exercise 4.16. ■

Problem 4.4 (Demography) We consider a population of a very large number M of individuals. The distribution $n(s)$ of their height can be represented by a "bell" function characterized by the mean value $\bar{h}$ of the height and the standard deviation σ

$$n(s) = \frac{M}{\sigma\sqrt{2\pi}}e^{-(s-\bar{h})^2/(2\sigma^2)}.$$

Then

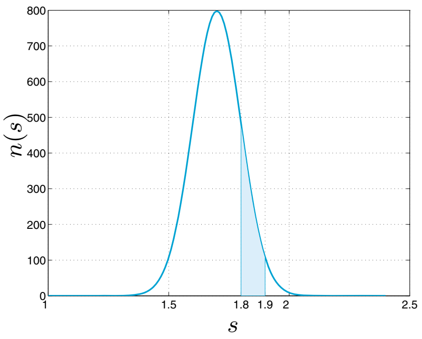


Figure 4.1. Height distribution of a population of $M = 200$ individuals

$$N_{[h,h+\varDelta h]} = \int_h^{h+\varDelta h} n(s)\;ds \tag{4.3}$$

represents the number of individuals whose height is between h and $h + \varDelta h$ (for a positive $\varDelta h$). An instance is provided in Figure 4.1, which corresponds to the case $M = 200$, $\bar{h} = 1.7$ m, $\sigma = 0.1$ m, and the area of the shadowed region gives the number of individuals whose height is in the range 1.8÷1.9 m. For the solution of this problem see Example 4.2. ■

4.2 Approximation of function derivatives

Consider a function $f : [a, b] \to \mathbb{R}$ continuously differentiable in $[a, b]$. We seek an approximation of the first derivative of f at a generic point $\bar{x}$ in (a, b).

In view of the definition (1.10), for h sufficiently small and positive, we can assume that the quantity

$$\boxed{(\delta_+ f)(\bar{x}) = \frac{f(\bar{x} + h) - f(\bar{x})}{h}} \tag{4.4}$$

is an approximation of $f'(\bar{x})$ which is called the *forward finite difference*. To estimate the error, it suffices to expand f in a Taylor series; if $f \in C^2((a, b))$, we have

$$f(\bar{x} + h) = f(\bar{x}) + hf'(\bar{x}) + \frac{h^2}{2} f''(\xi), \tag{4.5}$$

where ξ is a suitable point in the interval $(\bar{x}, \bar{x} + h)$. Therefore

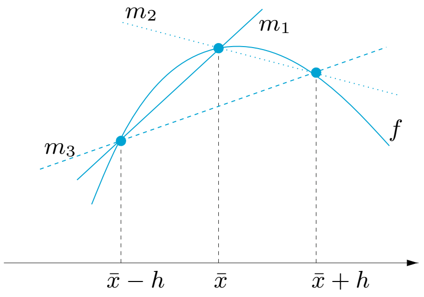


Figure 4.2. Finite difference approximation of $f'(\bar{x})$: backward (*solid line*), forward (*dotted line*) and centered (*dashed line*). The values $m_1 = (\delta_- f)(\bar{x})$, $m_2 = (\delta_+ f)(\bar{x})$ and $m_3 = (\delta f)(\bar{x})$ denote the slopes of the three straight lines

$$(\delta_+ f)(\bar{x}) = f'(\bar{x}) + \frac{h}{2} f''(\xi), \tag{4.6}$$

and thus $(\delta_+ f)(\bar{x})$ provides a first-order approximation to $f'(\bar{x})$ with respect to h. Still assuming $f \in C^2((a,b))$, with a similar procedure we can derive from the Taylor expansion

$$f(\bar{x}-h) = f(\bar{x}) - hf'(\bar{x}) + \frac{h^2}{2} f''(\eta) \tag{4.7}$$

with $\eta \in (\bar{x}-h, \bar{x})$, the *backward finite difference*

$$\boxed{(\delta_- f)(\bar{x}) = \frac{f(\bar{x}) - f(\bar{x}-h)}{h}} \tag{4.8}$$

which is also first-order accurate. Note that formulae (4.4) and (4.8) can also be obtained by differentiating the linear polynomial interpolating f at the points $\{\bar{x}, \bar{x}+h\}$ and $\{\bar{x}-h, \bar{x}\}$, respectively. In fact, these schemes amount to approximating $f'(\bar{x})$ by the slope of the straight line passing through the two points $(\bar{x}, f(\bar{x}))$ and $(\bar{x}+h, f(\bar{x}+h))$, or $(\bar{x}-h, f(\bar{x}-h))$ and $(\bar{x}, f(\bar{x}))$, respectively (see Figure 4.2).

Finally, we introduce the *centered finite difference* formula

$$\boxed{(\delta f)(\bar{x}) = \frac{f(\bar{x}+h) - f(\bar{x}-h)}{2h}} \tag{4.9}$$

If $f \in C^3((a,b))$, this formula provides a second-order approximation to $f'(\bar{x})$ with respect to h. Indeed, by expanding $f(\bar{x}+h)$ and $f(\bar{x}-h)$ at the third order around $\bar{x}$ and summing up the two expressions, we obtain

$$f'(\bar{x}) - (\delta f)(\bar{x}) = -\frac{h^2}{12}[f'''(\xi_-) + f'''(\xi_+)], \tag{4.10}$$

where ξ_- and ξ^+ are suitable points in the intervals $(\bar{x}-h, \bar{x})$ and $(\bar{x}, \bar{x}+h)$, respectively (see Exercise 4.2).

By (4.9) $f'(\bar{x})$ is approximated by the slope of the straight line passing through the points $(\bar{x}-h, f(\bar{x}-h))$ and $(\bar{x}+h, f(\bar{x}+h))$.

Example 4.1 (Hydraulics) Let us solve Problem 4.1, using formulae (4.4), (4.8) and (4.9), with $h = 5$, to approximate $q'(t)$ at five different points. We obtain:

t	0	5	10	15	20
$q'(t)$	-0.0212	-0.0194	-0.0176	-0.0159	-0.0141
$\delta_+ q$	-0.0203	-0.0185	-0.0168	-0.0150	$--$
$\delta_- q$	$--$	-0.0203	-0.0185	-0.0168	-0.0150
δq	$--$	-0.0194	-0.0176	-0.0159	$--$

The agreement between the exact derivative and the one computed from the finite difference formulae with $h = 5$ is more satisfactory when using formula (4.9) rather than (4.8) or (4.4). ■

In general, we can assume that the values of f are available at $n+1$ equispaced points $x_i = x_0 + ih$, $i = 0, \ldots, n$, with $h > 0$. In this case in the numerical derivation $f'(x_i)$ can be approximated by taking one of the previous formulae (4.4), (4.8) or (4.9) with $\bar{x} = x_i$.

Note that the centered formula (4.9) cannot be used at the extrema x_0 and x_n. For these nodes we could use the values

$$\begin{aligned} &\frac{1}{2h}\left[-3f(x_0) + 4f(x_1) - f(x_2)\right] \qquad \text{at } x_0, \\ &\frac{1}{2h}\left[3f(x_n) - 4f(x_{n-1}) + f(x_{n-2})\right] \text{ at } x_n, \end{aligned} \tag{4.11}$$

which are also second-order accurate with respect to h. They are obtained by computing at the point x_0 (respectively, x_n) the first derivative of the polynomial of degree 2 interpolating f at the nodes x_0, x_1, x_2 (respectively, x_{n-2}, x_{n-1}, x_n).

See Exercises 4.1-4.4.

4.3 Numerical integration

In this section we introduce numerical methods suitable for approximating the integral

$$I(f) = \int_a^b f(x)dx,$$

where f is an arbitrary continuous function in $[a, b]$. We start by introducing some simple formulae, which are indeed special instances of the family of Newton-Cotes formulae. Then we will introduce the so-called Gaussian formulae, that feature the highest possible degree of exactness for a given number of evaluations of the function f.

4.3.1 Midpoint formula

A simple procedure to approximate $I(f)$ can be devised by partitioning the interval $[a, b]$ into subintervals $I_k = [x_{k-1}, x_k]$, $k = 1, \ldots, M$, with $x_k = a + kH$, $k = 0, \ldots, M$ and $H = (b - a)/M$. Since

$$I(f) = \sum_{k=1}^{M} \int_{I_k} f(x)dx, \tag{4.12}$$

on each sub-interval I_k we can approximate the exact integral of f by that of a polynomial $\tilde{f}$ approximating f on I_k. The simplest solution consists in choosing $\tilde{f}$ as the constant polynomial interpolating f at the middle point of I_k:

$$\bar{x}_k = \frac{x_{k-1} + x_k}{2}.$$

In such a way we obtain the *composite midpoint quadrature formula*

$$\boxed{I^c_{mp}(f) = H \sum_{k=1}^{M} f(\bar{x}_k)} \tag{4.13}$$

The symbol *mp* stands for midpoint, while *c* stands for composite. This formula is second-order accurate with respect to H. More precisely, if f is continuously differentiable up to its second derivative in $[a, b]$, we have

$$I(f) - I^c_{mp}(f) = \frac{b-a}{24} H^2 f''(\xi), \tag{4.14}$$

where ξ is a suitable point in $[a, b]$ (see Exercise 4.6). Formula (4.13) is also called the *composite rectangle quadrature formula* because of its geometrical interpretation, which is evident from Figure 4.3.

The classical *midpoint formula* (or *rectangle formula*) is obtained by taking $M = 1$ in (4.13), i.e. using the midpoint rule directly on the interval (a, b):

$$\boxed{I_{mp}(f) = (b - a)f[(a + b)/2]} \tag{4.15}$$

The error is now given by

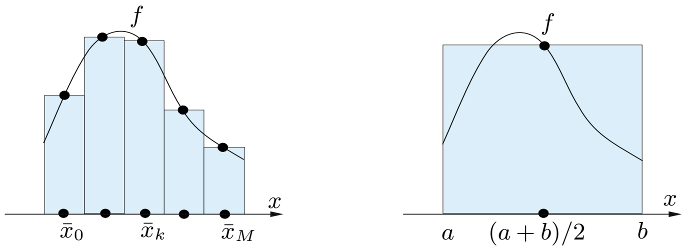


Figure 4.3. The composite midpoint formula (*left*); the midpoint formula (*right*)

$$I(f) - I_{mp}(f) = \frac{(b-a)^3}{24} f''(\xi), \tag{4.16}$$

where ξ is a suitable point in $[a, b]$. Relation (4.16) follows as a special case of (4.14), but it can also be proved directly. Indeed, setting $\bar{x} = (a+b)/2$, we have

$$\begin{aligned} I(f) - I_{mp}(f) &= \int_a^b [f(x) - f(\bar{x})]dx \\ &= \int_a^b f'(\bar{x})(x-\bar{x})dx + \frac{1}{2}\int_a^b f''(\eta(x))(x-\bar{x})^2 dx, \end{aligned}$$

where $\eta(x)$ is a suitable point in the interval whose endpoints are x and $\bar{x}$. Then (4.16) follows because $\int_a^b (x-\bar{x})dx = 0$ and, by the mean value theorem for integrals, there exists $\xi \in [a, b]$ such that

$$\frac{1}{2}\int_a^b f''(\eta(x))(x-\bar{x})^2 dx = \frac{1}{2} f''(\xi)\int_a^b (x-\bar{x})^2 dx = \frac{(b-a)^3}{24} f''(\xi).$$

The *degree of exactness* of a quadrature formula is the maximum integer $r \geq 0$ for which the approximate integral (produced by the quadrature formula) of any polynomial of degree r is equal to the exact integral. We can deduce from (4.14) and (4.16) that the midpoint formula has degree of exactness 1, since it integrates exactly all polynomials of degree less than or equal to 1 (but not all those of degree 2).

The midpoint composite quadrature formula is implemented in Program 4.1. Input parameters are the endpoints of the integration interval `a` and `b`, the number of subintervals `M` and the MATLAB function `f` to define the function f.

Program 4.1. midpointc: composite midpoint quadrature formula

```
function Imp=midpointc(a,b,M,fun,varargin)
%MIDPOINTC Composite midpoint numerical integration.
% IMP = MIDPOINTC(A,B,M,FUN) computes an approximation
% of the integral of the function FUN via the midpoint
% method (with M equal subintervals).  FUN accepts a
% real vector input x and returns a real vector value.
% FUN can be either an inline function, an anonymous
% function, or it can be defined by an external m-file.
% IMP=MIDPOINTC(A,B,M,FUN,P1,P2,...) calls the function
% FUN passing the optional parameters P1,P2,... as
% FUN(X,P1,P2,...).
H=(b-a)/M;
x = linspace(a+H/2,b-H/2,M);
fmp=fun(x,varargin{:}).*ones(1,M);
Imp=H*sum(fmp);
```

See the Exercises 4.5-4.8.

4.3.2 Trapezoidal formula

Another formula can be obtained by replacing f on I_k by the linear polynomial interpolating f at the nodes x_{k-1} and x_k (equivalently, replacing f by $\Pi_1^H f$, see Section 3.4, on the whole interval $[a, b]$). This yields

$$\boxed{\begin{aligned} I_t^c(f) &= \frac{H}{2}\sum_{k=1}^{M}[f(x_{k-1}) + f(x_k)] \\ &= \frac{H}{2}[f(a) + f(b)] + H\sum_{k=1}^{M-1} f(x_k) \end{aligned}} \tag{4.17}$$

This formula is called the *composite trapezoidal formula*, and is second-order accurate with respect to H. In fact, one can obtain the expression

$$I(f) - I_t^c(f) = -\frac{b-a}{12}H^2 f''(\xi) \tag{4.18}$$

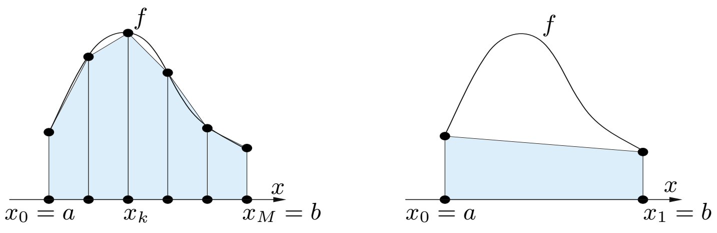


Figure 4.4. Composite trapezoidal formula (*left*); trapezoidal formula (*right*)

for the quadrature error for a suitable point $\xi \in [a,b]$, provided that $f \in C^2([a,b])$. When (4.17) is used with $M = 1$, we obtain

$$\boxed{I_t(f) = \frac{b-a}{2}[f(a) + f(b)]} \qquad (4.19)$$

which is called the *trapezoidal formula* because of its geometrical interpretation. The error induced is given by

$$I(f) - I_t(f) = -\frac{(b-a)^3}{12} f''(\xi), \qquad (4.20)$$

where ξ is a suitable point in $[a,b]$. We can deduce that (4.19) has degree of exactness equal to 1, as is the case of the midpoint rule.

The composite trapezoidal formula (4.17) is implemented in the MATLAB programs `trapz` and `cumtrapz`. If `x` is a vector whose components are the abscissae x_k, $k = 0,\ldots,M$ (with $x_0 = a$ and $x_M = b$), and `y` that of the values $f(x_k)$, $k = 0,\ldots,M$, `z=cumtrapz(x,y)` returns the vector `z` whose components are $z_k \simeq \int_a^{x_k} f(x)dx$, the integral being approximated by the composite trapezoidal rule. Thus `z(M+1)` is an approximation of the integral of f on (a,b).

`trapz`
`cumtrapz`

See the Exercises 4.9-4.11.

4.3.3 Simpson formula

The Simpson formula can be obtained by replacing the integral of f over each I_k by that of its interpolating polynomial of degree 2 at the nodes x_{k-1}, $\bar{x}_k = (x_{k-1} + x_k)/2$ and x_k,

$$\begin{aligned}\Pi_2 f(x) = {} & \frac{2(x-\bar{x}_k)(x-x_k)}{H^2} f(x_{k-1}) \\ & + \frac{4(x_{k-1}-x)(x-x_k)}{H^2} f(\bar{x}_k) + \frac{2(x-\bar{x}_k)(x-x_{k-1})}{H^2} f(x_k).\end{aligned}$$

The resulting formula is called the *composite Simpson quadrature formula*, and reads

$$\boxed{I_s^c(f) = \frac{H}{6}\sum_{k=1}^{M}[f(x_{k-1}) + 4f(\bar{x}_k) + f(x_k)]} \qquad (4.21)$$

One can prove that it induces the error

$$I(f) - I_s^c(f) = -\frac{b-a}{180}\frac{H^4}{16} f^{(4)}(\xi), \qquad (4.22)$$

where ξ is a suitable point in $[a,b]$, provided that $f \in C^4([a,b])$. It is therefore fourth-order accurate with respect to H. When (4.21) is applied to only one interval, say $[a,b]$, we obtain the so-called *Simpson quadrature formula*

$$\boxed{I_s(f) = \frac{b-a}{6}\left[f(a) + 4f((a+b)/2) + f(b)\right]} \tag{4.23}$$

The error is now given by

$$I(f) - I_s(f) = -\frac{1}{16}\frac{(b-a)^5}{180} f^{(4)}(\xi), \tag{4.24}$$

for a suitable $\xi \in [a,b]$. Its degree of exactness is therefore equal to 3.

The composite Simpson rule is implemented in Program 4.2.

Program 4.2. simpsonc: composite Simpson quadrature formula

```
function [Isic]=simpsonc(a,b,M,fun,varargin)
%SIMPSONC Composite Simpson numerical integration.
% ISIC = SIMPSONC(A,B,M,FUN) computes an approximation
% of the integral of the function FUN via the Simpson
% method (using M equal subintervals).  FUN accepts
% real vector input x and returns a real vector value.
% FUN can be either an inline function, an anonymous
% function, or it can be defined by an external m-file.
% ISIC = SIMPSONC(A,B,M,FUN,P1,P2,...) calls the
% function FUN passing the optional parameters
% P1,P2,... as FUN(X,P1,P2,...).
H=(b-a)/M;
x=linspace(a,b,M+1);
fpm=fun(x,varargin{:}).*ones(1,M+1);
fpm(2:end-1) = 2*fpm(2:end-1);
Isic=H*sum(fpm)/6;
x=linspace(a+H/2,b-H/2,M);
fpm=fun(x,varargin{:}).*ones(1,M);
Isic = Isic+2*H*sum(fpm)/3;
return
```

Example 4.2 (Demography) Let us consider Problem 4.4. To compute the number of individuals whose height is between 1.8 and 1.9 m, we need to solve the integral (4.3) for $h = 1.8$ and $\Delta h = 0.1$. For that we use the composite Simpson formula with 100 sub-intervals

```
M = 200; hbar = 1.7; sigma = 0.1;
N = @(h)M/(sigma*sqrt(2*pi))*exp(-(h-hbar).^...
    2./(2*sigma^2));
int = simpsonc(1.8, 1.9, 100, N)

int =
   27.1810
```

We therefore estimate that the number of individuals in this range of height is 27.1810, corresponding to the 15.39 % of all individuals. ■

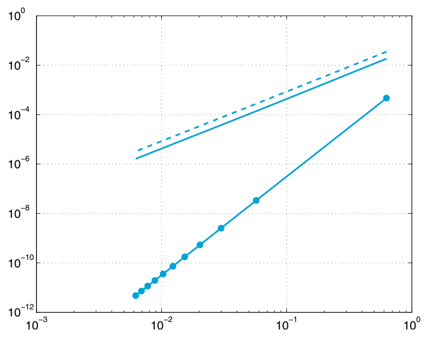


Figure 4.5. Logarithmic representation of the errors versus H for Simpson (*solid line with circles*), midpoint (*solid line*) and trapezoidal (*dashed line*) composite quadrature formulae

Example 4.3 We want to compare the approximations of the integral $I(f) = \int_0^{2\pi} xe^{-x}\cos(2x)dx = -(10\pi - 3 + 3e^{2\pi})/(25e^{2\pi}) \simeq -0.122122604618968$ obtained by using the composite midpoint, trapezoidal and Simpson formulae. In Figure 4.5 we plot on the logarithmic scale the errors versus H. As pointed out in Section 1.6, in this type of plot the greater the slope of the curve, the higher the order of convergence of the corresponding formula. As expected from the theoretical results, the midpoint and trapezoidal formulae are second-order accurate, whereas the Simpson formula is fourth-order accurate. ■

4.4 Interpolatory quadratures

Quadrature formulas like (4.15), (4.19) or (4.23), refer to a single interval, i.e. to $M = 1$, and for that they are said *simple* (or *non-composite*). They can be regarded as special instances of a more general quadrature formula of the form

$$\boxed{I_{appr}(f) = \sum_{j=0}^{n} \alpha_j f(y_j)} \tag{4.25}$$

The real numbers $\{\alpha_j\}$ are the *quadrature weights*, while the points $\{y_j\}$ are the *quadrature nodes*. In general, one requires that (4.25) integrates exactly at least a constant function: this property is ensured if $\sum_{j=0}^{n} \alpha_j = b - a$. We can get a degree of exactness equal to (at least) n taking

$$I_{appr}(f) = \int_a^b \Pi_n f(x)dx\,,$$

where $\Pi_n f \in \mathbb{P}_n$ is the Lagrange interpolating polynomial of the function f at the nodes $y_i, i = 0, \ldots, n$, given by (3.4). This yields the following expression for the weights

$$\alpha_i = \int_a^b \varphi_i(x)dx, \qquad i = 0, \ldots, n,$$

where $\varphi_i \in \mathbb{P}_n$ is the ith characteristic Lagrange polynomial such that $\varphi_i(y_j) = \delta_{ij}$, for $i, j = 0, \ldots, n$, that was introduced in (3.3).

Example 4.4 For the trapezoidal formula (4.19) we have $n = 1$, $y_0 = a$, $y_1 = b$ and

$$\alpha_0 = \int_a^b \varphi_0(x)dx = \int_a^b \frac{x-b}{a-b}dx = \frac{b-a}{2},$$
$$\alpha_1 = \int_a^b \varphi_1(x)dx = \int_a^b \frac{x-a}{b-a}dx = \frac{b-a}{2}.$$

■

The question that arises is whether suitable choices of the nodes exist such that the degree of exactness is greater than n, more precisely, equal to $r = n + m$ for some $m > 0$. We can simplify our discussion by restricting ourselves to a reference interval, say $[-1, 1]$. Indeed, once a set of quadrature nodes $\{\bar{y}_j\}$ and weights $\{\bar{\alpha}_j\}$ are available on $[-1, 1]$, then owing to the change of variable (3.11) we can immediately obtain the corresponding nodes and weights,

$$y_j = \frac{a+b}{2} + \frac{b-a}{2}\bar{y}_j, \qquad \alpha_j = \frac{b-a}{2}\bar{\alpha}_j$$

on an arbitrary integration interval $[a, b]$.

The answer to the previous question is furnished by the following result (see, [QSS07, Chapter 10]):

Proposition 4.1 *For a given $m > 0$, the quadrature formula $\sum_{j=0}^n \bar{\alpha}_j f(\bar{y}_j)$ has degree of exactness $n + m$ iff it is of interpolatory type and the nodal polynomial $\omega_{n+1} = \Pi_{i=0}^n (x - \bar{y}_i)$ associated with the nodes $\{\bar{y}_i\}$ is such that*

$$\int_{-1}^1 \omega_{n+1}(x)p(x)dx = 0, \qquad \forall p \in \mathbb{P}_{m-1}. \tag{4.26}$$

Table 4.1. Nodes and weights for some quadrature formulae of Gauss-Legendre type on the interval $[-1,1]$. Weights corresponding to symmetric couples of nodes are reported only once

n	$\{\bar{y}_j\}$	$\{\bar{\alpha}_j\}$
1	$\{\pm 1/\sqrt{3}\}$	$\{1\}$
2	$\{\pm\sqrt{15}/5, 0\}$	$\{5/9, 8/9\}$
3	$\{\pm(1/35)\sqrt{525-70\sqrt{30}},$	$\{(1/36)(18+\sqrt{30}),$
	$\pm(1/35)\sqrt{525+70\sqrt{30}}\}$	$(1/36)(18-\sqrt{30})\}$
4	$\{0, \pm(1/21)\sqrt{245-14\sqrt{70}}$	$\{128/225, (1/900)(322+13\sqrt{70})$
	$\pm(1/21)\sqrt{245+14\sqrt{70}}\}$	$(1/900)(322-13\sqrt{70})\}$

The maximum value that m can take is $n+1$ and is achieved provided ω_{n+1} is proportional to the so-called Legendre polynomial of degree $n+1$, $L_{n+1}(x)$. The Legendre polynomials can be computed recursively, through the following three-term relation

$$\begin{aligned} &L_0(x) = 1, \qquad L_1(x) = x, \\ &L_{k+1}(x) = \frac{2k+1}{k+1} x L_k(x) - \frac{k}{k+1} L_{k-1}(x), \qquad k = 1, 2, \ldots. \end{aligned}$$

For every $n = 0, 1, \ldots$, every polynomial $p_n \in \mathbb{P}_n$ can be obtained by a linear combination of the polynomials $L_0, L_1, \ldots, L_n$. Moreover, L_{n+1} is orthogonal to all the Legendre polynomials of degree less than or equal to n, i.e., $\int_{-1}^{1} L_{n+1}(x) L_j(x) dx = 0$ for all $j = 0, \ldots, n$. This explains why (4.26) is true with m less than or equal to $n+1$.

The maximum degree of exactness is therefore equal to $2n+1$, and is obtained for the so-called *Gauss-Legendre formula* (I_{GL} in short), whose nodes and weights are given by:

$$\begin{cases} \bar{y}_j = \text{ zeros of } L_{n+1}(x), \\ \bar{\alpha}_j = \dfrac{2}{(1-\bar{y}_j^2)[L'_{n+1}(\bar{y}_j)]^2}, \qquad j = 0, \ldots, n. \end{cases} \tag{4.27}$$

The weights $\bar{\alpha}_j$ are all positive and the nodes are internal to the interval $[-1,1]$. In Table 4.1 we report nodes and weights for the Gauss-Legendre quadrature formulae with $n = 1, 2, 3, 4$. If $f \in C^{(2n+2)}([-1,1])$, the corresponding error is

$$I(f) - I_{GL}(f) = \frac{2^{2n+3}((n+1)!)^4}{(2n+3)((2n+2)!)^3} f^{(2n+2)}(\xi),$$

where ξ is a suitable point in $(-1,1)$.

It is often useful to include also the endpoints of the interval among the quadrature nodes. By doing so, the Gauss formula with the highest

Table 4.2. Nodes and weights for some quadrature formulae of Gauss-Legendre-Lobatto on the interval $[-1, 1]$. Weights corresponding to symmetric couples of nodes are reported only once

n	$\{\bar{y}_j\}$	$\{\bar{\alpha}_j\}$
1	$\{\pm 1\}$	$\{1\}$
2	$\{\pm 1, 0\}$	$\{1/3, 4/3\}$
3	$\{\pm 1, \pm\sqrt{5}/5\}$	$\{1/6, 5/6\}$
4	$\{\pm 1, \pm\sqrt{21}/7, 0\}$	$\{1/10, 49/90, 32/45\}$

degree of exactness $(2n-1)$ is the one that employs the so-called *Gauss-Legendre-Lobatto* nodes (briefly, GLL): for $n \geq 1$

$$\overline{y}_0 = -1,\ \overline{y}_n = 1, \overline{y}_j = \text{zeros of } L'_n(x), \quad j = 1, \dots, n-1, \quad (4.28)$$

$$\overline{\alpha}_j = \frac{2}{n(n+1)} \frac{1}{[L_n(\overline{y}_j)]^2}, \qquad j = 0, \dots, n.$$

If $f \in C^{(2n)}([-1, 1])$, the corresponding error is given by

$$I(f) - I_{GLL}(f) = -\frac{(n+1)n^3 2^{2n+1}((n-1)!)^4}{(2n+1)((2n)!)^3} f^{(2n)}(\xi),$$

for a suitable $\xi \in (-1, 1)$. In Table 4.2 we give a table of nodes and weights on the reference interval $[-1, 1]$ for $n = 1, 2, 3, 4$. (For $n = 1$ we recover the trapezoidal rule.)

quadl Using the MATLAB instruction `quadl(fun,a,b)` it is possible to compute an integral with a composite Gauss-Legendre-Lobatto quadrature formula. The input arguments are: the function handle `fun` associated with the function f, the endpoints `a` and `b` of the integration interval. For instance, to integrate $f(x) = 1/x$ over $[1, 2]$, we must first define the function

```
fun=@(x) 1./x;
```

then call `quadl(fun,1,2)`. Note that in the definition of function f we have used an element by element operation (indeed MATLAB will evaluate this expression component by component on the vector of quadrature nodes).

The specification of the number of subintervals is not requested as it is automatically computed in order to ensure that the quadrature error is below the default tolerance of 10^{-3}. A different tolerance can be provided by the user through the extended command `quadl(fun,a,b,tol)`. In Section 4.5 we will introduce a method to estimate the quadrature error and, consequently, to change H adaptively.

Let us summarize

1. A quadrature formula is a formula to approximate the integral of continuous functions on an interval $[a, b]$;
2. it is generally expressed as a linear combination of the values of the function at specific points (called *nodes*) with coefficients which are called *weights*;
3. the *degree of exactness* of a quadrature formula is the highest degree of the polynomials which are integrated exactly by the formula. It is one for the midpoint and trapezoidal rules, three for the Simpson rule, $2n + 1$ for the Gauss-Legendre formula using $n + 1$ quadrature nodes, and $2n - 1$ for the Gauss-Legendre-Lobatto formula using $n + 1$ nodes;
4. the *order of accuracy* of a composite quadrature formula is its order with respect to the size H of the subintervals. The order of accuracy is two for composite midpoint and trapezoidal formulae, four for composite Simpson formula.

See the Exercises 4.12-4.18.

4.5 Simpson adaptive formula

The integration steplength H of a composite quadrature formula (4.21) can be chosen in order to ensure that the quadrature error is less than a prescribed tolerance $\varepsilon > 0$. For instance, when using the Simpson composite formula, thanks to (4.22) this goal can be achieved if

$$\frac{b-a}{180}\frac{H^4}{16}\max_{x\in[a,b]}|f^{(4)}(x)| < \varepsilon, \tag{4.29}$$

where $f^{(4)}$ denotes the fourth-order derivative of f. Unfortunately, when the absolute value of $f^{(4)}$ is large only in a small part of the integration interval, the maximum H for which (4.29) holds true can be too small. The goal of the adaptive Simpson quadrature formula is to yield an approximation of $I(f)$ within a fixed tolerance ε by a *nonuniform* distribution of the integration steplengths in the interval $[a, b]$. In such a way we retain the same accuracy of the composite Simpson rule, but with a lower number of quadrature nodes and, consequently, a reduced number of evaluations of f.

To this end, we must find an error estimator and an automatic procedure to modify the integration steplength H, according to the achievement of the prescribed tolerance. We start by analyzing this procedure, which is independent of the specific quadrature formula that one wants to apply.

In the first step of the adaptive procedure, we compute an approximation $I_s(f)$ of $I(f) = \int_a^b f(x)dx$. We set $H = b - a$ and we try to estimate the quadrature error. If the error is less than the prescribed tolerance, the adaptive procedure is stopped; otherwise the steplength H is halved until the integral $\int_a^{a+H} f(x)dx$ is computed with the prescribed accuracy. When the test is passed, we consider the interval $(a + H, b)$ and we repeat the previous procedure, choosing as the first steplength the length $b - (a + H)$ of that interval.

We use the following notations:

1. A: the *active* integration interval, i.e. the interval where the integral is being computed;
2. S: the integration interval already examined, for which the error is less than the prescribed tolerance;
3. N: the integration interval yet to be examined.

At the beginning of the integration process we have $A = [a, b]$, $N = \emptyset$ and $S = \emptyset$, the situation at the generic step of the algorithm is depicted in Figure 4.6. Let $J_S(f)$ indicate the computed approximation of $\int_a^\alpha f(x)dx$, with $J_S(f) = 0$ at the beginning of the process; if the algorithm successfully terminates, $J_S(f)$ yields the desired approximation of $I(f)$. We also denote by $J_{(\alpha,\beta)}(f)$ the approximate integral of f over the active interval $[\alpha, \beta]$. This interval is drawn in white in Figure 4.6. The generic step of the adaptive integration method is organized as follows:

1. if the estimation of the error ensures that the prescribed tolerance is satisfied, then:
   (i) $J_S(f)$ is increased by $J_{(\alpha,\beta)}(f)$, that is $J_S(f) \leftarrow J_S(f) + J_{(\alpha,\beta)}(f)$;
   (ii) we let $S \leftarrow S \cup A$, $A = N$, $N = \emptyset$ (corresponding to the path (I) in Figure 4.6) and $\alpha \leftarrow \beta$ and $\beta \leftarrow b$;
2. if the estimation of the error fails the prescribed tolerance, then:
   (j) A is halved, and the new active interval is set to $A = [\alpha, \alpha']$ with $\alpha' = (\alpha + \beta)/2$ (corresponding to the path (II) in Figure 4.6);
   (jj) we let $N \leftarrow N \cup [\alpha', \beta]$, $\beta \leftarrow \alpha'$;
   (jjj) a new error estimate is provided.

Of course, in order to prevent the algorithm from generating too small steplengths, it is convenient to monitor the width of A and warn the user, in case of an excessive reduction of the steplength, about the presence of a possible singularity in the integrand function.

The problem now is to find a suitable estimator of the error. To this end, it is convenient to restrict our attention to a generic subinterval $[\alpha, \beta] \subset [a, b]$ in which we compute $I_s(f)$: of course, if on this interval

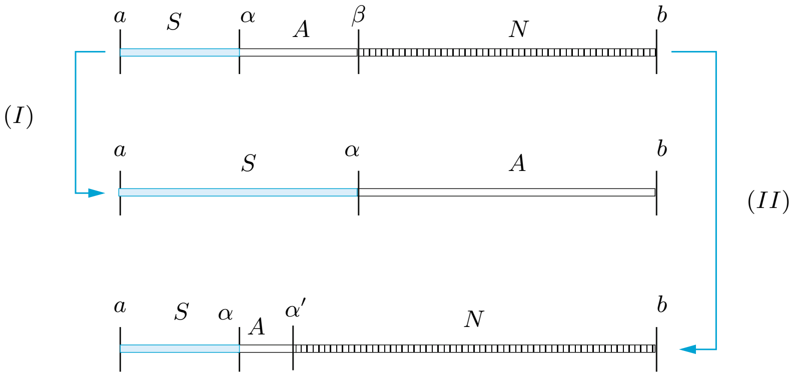


Figure 4.6. Distribution of the integration intervals at the generic step of the adaptive algorithm and updating of the integration grid

the error is less than $\varepsilon(\beta-\alpha)/(b-a)$, then the error on the interval $[a,b]$ will be less than the prescribed tolerance ε. Since from (4.24) we get

$$E_s(f;\alpha,\beta) = \int_\alpha^\beta f(x)dx - I_s(f) = -\frac{(\beta-\alpha)^5}{2880} f^{(4)}(\xi),$$

to ensure the achievement of the tolerance, it will be sufficient to verify that $E_s(f;\alpha,\beta) < \varepsilon(\beta-\alpha)/(b-a)$. In practical computation, this procedure is not feasible since the point $\xi \in [\alpha,\beta]$ is unknown.

To estimate the error $E_s(f;\alpha,\beta)$ without using explicitly the value $f^{(4)}(\xi)$, we employ again the composite Simpson formula to compute $\int_\alpha^\beta f(x)dx$, but with a steplength $H = (\beta-\alpha)/2$. From (4.22) with $a=\alpha$ and $b=\beta$, we deduce that

$$\int_\alpha^\beta f(x)dx - I_s^c(f) = -\frac{(\beta-\alpha)^5}{46080} f^{(4)}(\eta), \tag{4.30}$$

where η is a suitable point different from ξ. Subtracting the last two equations, we get

$$\varDelta I = I_s^c(f) - I_s(f) = -\frac{(\beta-\alpha)^5}{2880} f^{(4)}(\xi) + \frac{(\beta-\alpha)^5}{46080} f^{(4)}(\eta). \tag{4.31}$$

Let us now make the assumption that $f^{(4)}(x)$ is approximately a constant on the interval $[\alpha,\beta]$. In this case $f^{(4)}(\xi) \simeq f^{(4)}(\eta)$. We can compute $f^{(4)}(\eta)$ from (4.31) and, putting this value in the equation (4.30), we obtain the following estimation of the error:

$$\int_\alpha^\beta f(x)dx - I_s^c(f) \simeq \frac{1}{15}\varDelta I.$$

The steplength $(\beta-\alpha)/2$ (that is the steplength employed to compute $I_s^c(f)$) will be accepted if $|\Delta I|/15 < \varepsilon(\beta-\alpha)/[2(b-a)]$. The quadrature formula that uses this criterion in the adaptive procedure described previously, is called *adaptive Simpson formula*. It is implemented in Program 4.3. Among the input parameters, `f` is the string in which the function f is defined, `a` and `b` are the endpoints of the integration interval, `tol` is the prescribed tolerance on the error and `hmin` is the minimum admissible value for the integration steplength (in order to ensure that the adaptation procedure always terminates).

Program 4.3. simpadpt: adaptive Simpson formula

```
function [JSf,nodes]=simpadpt(fun,a,b,tol,hmin,varargin)
%SIMPADPT Adaptive Simpson quadrature formula
% JSF = SIMPADPT(FUN,A,B,TOL,HMIN) tries to approximate
% the integral of function FUN from A to B within
% error TOL using recursive adaptive Simpson
% quadrature with H>=HMIN. The function FUN should
% accept a vector argument x and return a vector.
% FUN can be either an inline function, an anonymous
% function, or it can be defined by an external m-file.
% JSF = SIMPADPT(FUN,A,B,TOL,HMIN,P1,P2,...) calls the
% function FUN passing the optional parameters
% P1,P2,... as FUN(X,P1,P2,...).
% [JSF,NODES] = SIMPADPT(...) returns the distribution
% of nodes used in the quadrature process.
A=[a,b]; N=[]; S=[]; JSf = 0; ba = 2*(b - a); nodes=[];
while ~isempty(A),
  [deltaI,ISc]=caldeltai(A,fun,varargin{:});
  if abs(deltaI) < 15*tol*(A(2)-A(1))/ba;
     JSf = JSf + ISc;    S = union(S,A);
     nodes = [nodes, A(1) (A(1)+A(2))*0.5 A(2)];
     S = [S(1), S(end)]; A = N; N = [];
  elseif A(2)-A(1) < hmin
     JSf=JSf+ISc;        S = union(S,A);
     S = [S(1), S(end)]; A=N; N=[];
     warning('Too small integration-step');
  else
     Am = (A(1)+A(2))*0.5;
     A = [A(1) Am];   N = [Am, b];
  end
end
nodes=unique(nodes);
return

function [deltaI,ISc]=caldeltai(A,fun,varargin)
L=A(2)-A(1);
t=[0; 0.25; 0.5; 0.75; 1];
x=L*t+A(1); L=L/6;
w=[1; 4; 1]; wp=[1;4;2;4;1];
fx=fun(x,varargin{:}).*ones(5,1);
IS=L*sum(fx([1 3 5]).*w);
ISc=0.5*L*sum(fx.*wp);
deltaI=IS-ISc;
return
```

Example 4.5 Let us compute the integral $I(f) = \int_{-1}^{1} 20(1-x^2)^3 dx$ by using the adaptive Simpson formula. Using Program 4.3 with

```
fun=@(x)(1-x.^2).^3*20;
tol = 1.e-04; hmin = 1.e-03; a=-1;b=1;
```

we find the approximate value 18.2857116732797, instead of the exact value 18.2857142857143. The error is less than the prescribed tolerance `tol`$=10^{-4}$, precisely it is 2.6124 10^{-6}. To obtain this result the adaptive formula requires 41 function evaluations; the corresponding composite formula with uniform steplength would have required 90 function evaluations to yield an error of 2.5989 10^{-6}. ■

4.6 Monte Carlo Methods for Numerical Integration

Monte Carlo methods are nowadays widely used for the solution of stochastic differential equations and uncertainty quantification problems.

Numerical integration methods based on Monte Carlo techniques are a valid tool for approximating multidimensional integrals when the space dimension of $\mathbb{R}^n$ gets large. These methods differ from the approaches considered thus far, since the choice of quadrature nodes is done *statistically* according to the values attained by random variables having a known probability distribution.

We recall that a *random* (or *stochastic*) *variable* $X = X(\zeta) = (X_1(\zeta), \ldots, X_n(\zeta)) \in \mathbb{R}^n$ (or, more properly, *random vector*) is a function defined for any outcome ζ of a random experiment, such that there exists a *probability density function* $p(X)$ associated with it (see [Pap87], Chapter 4).

The *probability density function* p associated with the random variable $X \in \mathbb{R}^n$ is a real valued function satisfying

$$p(X_1, \ldots, X_n) \geq 0, \qquad \int_{\mathbb{R}^n} p(X_1, \ldots, X_n) dX = 1.$$

For a vector $\mathbf{x} = (x_1, \ldots, x_n)^T \in \mathbb{R}^n$, the *probability* $\mathcal{P}\{X \leq \mathbf{x}\}$ of the random event $\{X_1 \leq x_1, \ldots, X_n \leq x_n\}$ is given by

$$\mathcal{P}\{X \leq \mathbf{x}\} = \int_{-\infty}^{x_n} \ldots \int_{-\infty}^{x_1} p(X_1, \ldots, X_n) dX_1 \ldots dX_n.$$

Finally, given a function f defined on the random variable X with associated probability density function $p(X)$, the *statistical mean* (or *expectation*) of f is

$$\mu(f) = \int_{\mathbb{R}^n} f(X) p(X) dX. \tag{4.32}$$

The basic idea of Monte Carlo method is to interpret the integral of a function f in terms of the *statistical mean* of the function itself. Identifying the generic point $\mathbf{x} = (x_1, x_2, \ldots, x_n)^T \in \mathbb{R}^n$ with the random variable $X = (X_1, X_2, \ldots, X_n)^T$ we have

$$\int_{\Omega} f(\mathbf{x})d\mathbf{x} = |\Omega| \int_{\mathbb{R}^n} |\Omega|^{-1} \chi_{\Omega}(X) f(X) dX = |\Omega| \mu(f),$$

where:

- $|\Omega|$ denotes the n-dimensional measure of Ω,
- $\chi_{\Omega}(X)$ is the characteristic function of the set Ω, equal to 1 for $X \in \Omega$ and to 0 elsewhere,
- $p(X) = |\Omega|^{-1}\chi_{\Omega}(X)$ is the probability density function associated with the random variable X.

The numerical computation of the mean value $\mu(f)$ is carried out by taking N mutually independent samples $X_1, \ldots, X_N \in \mathbb{R}^n$ of the random variable X and evaluating the *average*

$$I_N(f) = \frac{1}{N}\sum_{i=1}^{N} f(X_i). \tag{4.33}$$

The strong law of large numbers ensures with probability 1 the convergence of the average $I_N(f)$ to the mean value $\mu(f)$ as $N \to \infty$. In computational practice the sequence of samples $X_1, \ldots, X_N$ is deterministically produced by a random-number generator, giving rise to the so-called *pseudo-random integration formulae.*

The quadrature error $E_N(f) = \mu(f) - I_N(f)$, as a function of N, can be characterized through the *variance*

$$\sigma^2(I_N(f)) = \mu\left((\mu(f) - I_N(f))^2\right).$$

Moreover, it holds

$$\sigma^2(I_N(f)) = \frac{\sigma^2(f)}{N}, \tag{4.34}$$

from which, as $N \to \infty$, a convergence rate of $\mathcal{O}(N^{-1/2})$ follows for the statistical estimate of the error $E_N(f)$, as $E_N(f) \propto \sqrt{\sigma^2(I_N(f))}$ (see, e.g. [KW08]).

Such convergence rate *does not* depend on the dimension n of the integration domain, and this is a most relevant feature of the Monte Carlo method. However, it is worth noting that the convergence rate is independent of the *regularity* of f; thus, unlike interpolatory quadratures, Monte Carlo methods do not yield more accurate results when dealing with smooth integrands.

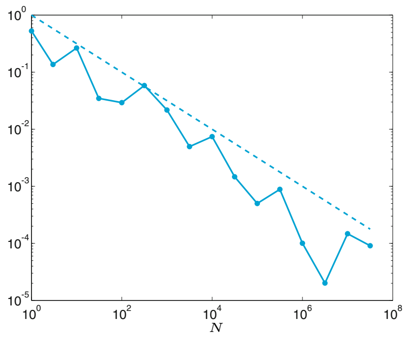


Figure 4.7. Absolute values of the errors $E_N(f)$ of Monte Carlo method (continuous line) and $N^{-1/2}$ (dashed line) for the functions of Example 4.6. At left, the 2D case. At right, the 3D case.

The estimate (4.34) is extremely weak and in practice one does often obtain poorly accurate results. A more efficient implementation of Monte Carlo methods is based on composite approach or semi-analytical methods; an example of these techniques is provided in the NAG (Numerical Algorithms Group) Library, where a composite Monte Carlo method is employed for the computation of integrals over hypercubes in $\mathbb{R}^n$.

Example 4.6 Let us compute the integrals $I_{2D} = \int_{[0,1]^2} \sin(\pi x)\cos(\pi y)dxdy$ and $I_{3D} = \int_{[0,1]^3} \sin(\pi x)\cos(\pi y)\sin(\pi z)dxdydz$ by Monte Carlo formula (4.33) with $N = 10^{k/2}$ and $k = 0, \ldots, 15$. The absolute value of the errors $E_N(f)$ are plotted in Figure 4.7 (at left the 2D case, at right the 3D case). The convergence rate $\mathcal{O}(N^{1/2})$ of the error is observed in both cases. ■

4.7 What we haven't told you

The midpoint, trapezoidal and Simpson formulae are particular cases of a larger family of quadrature rules known as *Newton-Cotes formulae*. For an introduction, see [QSS07, Chapter 9]. Similarly, the Gauss-Legendre and the Gauss-Legendre-Lobatto formulae that we have introduced in Section 4.4 are special cases of a more general family of Gaussian quadrature formulae. These are *optimal* in the sense that they maximize the degree of exactness for a prescribed number of quadrature nodes. For an introduction to Gaussian formulae, see [QSS07, Chapter 10] or [RR01]. Further developments on numerical integration can be found, e.g., in [DR75] and [PdDKÜK83].

Numerical integration can also be used to compute integrals on unbounded intervals. For instance, to approximate $\int_0^\infty f(x)dx$, a first possibility is to find a point α such that the value of $\int_\alpha^\infty f(x)dx$ can be neglected with respect to that of $\int_0^\alpha f(x)dx$. Then we compute by a quadrature formula this latter integral on a bounded interval. A second possibility is to resort to Gaussian quadrature formulae for unbounded intervals (see [QSS07, Chapter 10]).

Finally, numerical integration can also be used to compute multidimensional integrals. In particular, we mention the MATLAB instruction `dblquad(fun,xmin,xmax,ymin,ymax)` by which it is possible to compute the integral of a given function $f(x,y)$ on the rectangular domain $[xmin, xmax] \times [ymin, ymax]$. The function to be integrated is defined through the function handle `fun` that must be defined on two variables (e.g., x and y). If the function f is defined in the MATLAB file `fun.m`, the calling instruction is:
`dblquad(@fun,xmin,xmax,ymin,ymax)`.

dblquad

4.8 Exercises

Exercise 4.1 Verify that, if $f \in C^3$ in a neighborhood I_0 of x_0 (respectively, I_n of x_n) the error of formula (4.11) is equal to $-\frac{1}{3}f'''(\xi_0)h^2$ (respectively, $-\frac{1}{3}f'''(\xi_n)h^2$), where ξ_0 and ξ_n are two suitable points belonging to I_0 and I_n, respectively.

Exercise 4.2 Verify that if $f \in C^3$ in a neighborhood of $\bar{x}$ the error of the formula (4.9) is equal to (4.10).

Exercise 4.3 Compute the order of accuracy with respect to h of the following formulae for the numerical approximation of $f'(x_i)$:

$$a. \quad \frac{-11f(x_i) + 18f(x_{i+1}) - 9f(x_{i+2}) + 2f(x_{i+3})}{6h},$$

$$b. \quad \frac{f(x_{i-2}) - 6f(x_{i-1}) + 3f(x_i) + 2f(x_{i+1})}{6h},$$

$$c. \quad \frac{-f(x_{i-2}) - 12f(x_i) + 16f(x_{i+1}) - 3f(x_{i+2})}{12h}.$$

Exercise 4.4 (Demography) The following values represent the time evolution of the number $n(t)$ of individuals of a given population whose birth rate is constant ($b = 2$) and mortality rate is $d(t) = 0.01n(t)$:

t (months)	0	0.5	1	1.5	2	2.5	3
n	100	147	178	192	197	199	200

Use this data to approximate as accurately as possible the rate of variation of this population. Then compare the obtained results with the exact rate $n'(t) = 2n(t) - 0.01n^2(t)$.

Exercise 4.5 Find the minimum number M of subintervals to approximate with an absolute error less than 10^{-4} the integrals of the following functions:

$$\begin{aligned} f_1(x) &= \frac{1}{1+(x-\pi)^2} && \text{in } [0,5], \\ f_2(x) &= e^x\cos(x) && \text{in } [0,\pi], \\ f_3(x) &= \sqrt{x(1-x)} && \text{in } [0,1], \end{aligned}$$

using the composite midpoint formula. Verify the results obtained using the Program 4.1.

Exercise 4.6 Prove (4.14) starting from (4.16).

Exercise 4.7 Why does the midpoint formula lose one order of convergence when used in its composite mode?

Exercise 4.8 Verify that, if f is a polynomial of degree less than or equal 1, then $I_{mp}(f) = I(f)$ i.e. the midpoint formula has degree of exactness equal to 1.

Exercise 4.9 For the function f_1 in Exercise 4.5, compute (numerically) the values of M which ensure that the quadrature error is less than 10^{-4} when the integral is approximated by the composite trapezoidal and composite Gauss-Legendre (with $n = 1$) quadrature formulae.

Exercise 4.10 Let I_1 and I_2 be two values obtained by the composite trapezoidal formula applied with two different steplengths, H_1 and H_2, for the approximation of $I(f) = \int_a^b f(x)dx$. Verify that, if $f^{(2)}$ has a mild variation on (a,b), the value

$$I_R = I_1 + (I_1 - I_2)/(H_2^2/H_1^2 - 1) \tag{4.35}$$

is a better approximation of $I(f)$ than I_1 and I_2. This strategy is called the *Richardson extrapolation method*. Derive (4.35) from (4.18).

Exercise 4.11 Verify that, among all formulae of the form $I_{appr}(f) = \alpha f(\bar{x}) + \beta f(\bar{z})$ where $\bar{x}, \bar{z} \in [a,b]$ are two unknown nodes and α and β two undetermined weights, the Gauss formula with $n = 1$ of Table 4.1 features the maximum degree of exactness.

Exercise 4.12 For the first two functions of Exercise 4.5, compute the minimum number of intervals such that the quadrature error of the composite Simpson quadrature formula is less than 10^{-4}.

Exercise 4.13 Compute $\int_0^2 e^{-x^2/2}dx$ using the Simpson formula (4.23) and the Gauss-Legendre formula of Table 4.1 in the case $n = 1$, then compare the obtained results.

Exercise 4.14 To compute the integrals $I_k = \int_0^1 x^k e^{x-1} dx$ for $k = 1, 2, \ldots$, one can use the following recursive formula: $I_k = 1 - kI_{k-1}$, with $I_1 = 1/e$. Compute I_{20} using the composite Simpson formula in order to ensure that the quadrature error is less than 10^{-3}. Compare the Simpson approximation with the result obtained using the above recursive formula.

Exercise 4.15 Derive the Richardson extrapolation method for both Simpson formula (4.23) and Gauss-Legendre formula of Table 4.1 for $n = 1$. Then apply it for the approximation of the integral $I(f) = \int_0^2 e^{-x^2/2} dx$, with $H_1 = 1$ and $H_2 = 0.5$. Verify that in both cases I_R is more accurate than I_1 and I_2.

Exercise 4.16 (Electromagnetism) Compute using the composite Simpson formula the function $j(r, 0)$ defined in (4.2) for $r = k/10$ m with $k = 1, \ldots, 10$, $\rho(\xi) = e^{\xi}$ and $\sigma = 0.36$ W/(mK). Ensure that the quadrature error is less than 10^{-10}. (Recall that: m=meters, W=watts, K=degrees Kelvin.)

Exercise 4.17 (Optics) By using the composite Simpson and Gauss-Legendre with $n = 1$ formulae compute the function $E(T)$, defined in (4.1), for T equal to 213 K, up to at least 10 exact significant digits.

Exercise 4.18 Develop a strategy to compute $I(f) = \int_0^1 |x^2 - 0.25| dx$ by the composite Simpson formula such that the quadrature error is less than 10^{-2}.

5

Linear systems

In applied sciences, one is quite often led to face a linear system of the form

$$\mathrm{A}\mathbf{x} = \mathbf{b}, \tag{5.1}$$

where A is a square matrix of dimension $n \times n$ whose elements a_{ij} are either real or complex, while $\mathbf{x}$ and $\mathbf{b}$ are column vectors of dimension n: $\mathbf{x}$ represents the unknown solution while $\mathbf{b}$ is a given vector. Componentwise, (5.1) can be written as

$$\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1,\\
a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n &= b_2,\\
\vdots \qquad\qquad\qquad\quad \vdots \quad &\quad \vdots\\
a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n &= b_n.
\end{aligned}$$

Before proceeding we present four different problems that give rise to linear systems.

5.1 Some representative problems

Problem 5.1 (Hydraulic network) Let us consider the hydraulic network made of the 10 pipelines in Figure 5.1, which is fed by a reservoir of water at constant pressure $p_0 = 10$ bar. In this problem, pressure values refer to the difference between the real pressure and the atmospheric one. For the jth pipeline, the following relationship holds between the flow-rate Q_j (in m^3/s) and the pressure gap Δp_j at pipe-ends

$$Q_j = \frac{1}{R_j L_j}\Delta p_j, \tag{5.2}$$

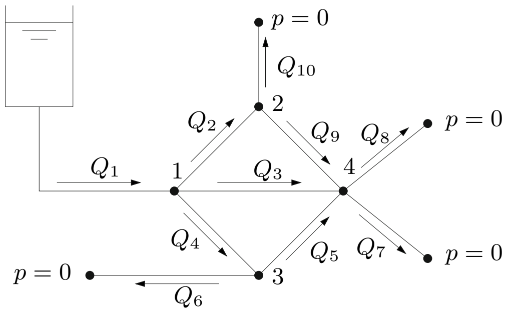


Figure 5.1. The pipeline network of Problem 5.1

where R_j is the hydraulic resistance per unit length (in (bar s)/m^4) and L_j is the length (in m) of the jth pipeline. We assume that water flows from the outlets (indicated by a black dot) at atmospheric pressure, which is set to 0 bar for coherence with the previous convention.

A typical problem consists in determining the pressure values at each internal node 1, 2, 3, 4. With this aim, for each $j = 1, 2, 3, 4$ we can supplement the relationship (5.2) with the statement that the algebraic sum of the flow-rates of the pipelines which meet at node j must be null (a negative value would indicate the presence of a seepage).

Denoting by $\mathbf{p} = (p_1, p_2, p_3, p_4)^T$ the pressure vector at the internal nodes, we get a 4×4 system of the form $\mathrm{A}\mathbf{p} = \mathbf{b}$.

In the following table we report the relevant characteristics of the different pipelines identified by the index j:

j	R_j	L_j	j	R_j	L_j	j	R_j	L_j
1	0.2500	20	2	2.0000	10	3	1.0204	14
4	2.0000	10	5	2.0000	10	6	7.8125	8
7	7.8125	8	8	7.8125	8	9	2.0000	10
10	7.8125	8						

Correspondingly, A and $\mathbf{b}$ take the following values (only the first 4 significant digits are provided):

$$\mathrm{A} = \begin{bmatrix} -0.370 & 0.050 & 0.050 & 0.070 \\ 0.050 & -0.116 & 0 & 0.050 \\ 0.050 & 0 & -0.116 & 0.050 \\ 0.070 & 0.050 & 0.050 & -0.202 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} -2 \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

The solution of this system is postponed to Example 5.5. ■

Problem 5.2 (Spectrometry) Let us consider a gas mixture of n non-reactive unknown components. Using a mass spectrometer the compound is bombarded by low-energy electrons: the resulting mixture of ions is analyzed by a galvanometer which shows peaks corresponding to specific ratios mass/charge. We only consider the n most relevant peaks. One may conjecture that the height h_i of the ith peak is a linear combination of $\{p_j, j = 1, \ldots, n\}$, p_j being the partial pressure of the jth component (that is the pressure exerted by a single gas when it is part of a mixture), yielding

$$\sum_{j=1}^{n} s_{ij} p_j = h_i, \qquad i = 1, \ldots, n, \tag{5.3}$$

where the s_{ij} are the so-called sensitivity coefficients. The determination of the partial pressures demands therefore the solution of a linear system. For its solution, see Example 5.3. ■

Problem 5.3 (Economy: input-output analysis) We want to determine the situation of equilibrium between demand and offer of certain goods. In particular, let us consider a production model in which $m \geq n$ factories (or production lines) produce n different products. They must face the internal demand of goods (the input) necessary to the factories for their own production, as well as the external demand (the output) from the consumers. Leontief proposed in (1930)[1] the amount of a certain output is proportional to the quantity of input used. Under this assumption the activity of the factories is completely described by two matrices, the input matrix C= $(c_{ij}) \in \mathbb{R}^{n\times m}$ and the output matrix P= $(p_{ij}) \in \mathbb{R}^{n\times m}$. ("C" stands for *consumables* and "P" for *products.*) The coefficient c_{ij} (respectively, p_{ij}) represent the quantity of the ith good absorbed (respectively, produced) by the jth factory for a fixed period of time. The matrix A=P−C is called *input-output matrix*: a positive (resp., negative) entry a_{ij} denotes the quantity of the ith good produced (respectively, absorbed) by the jth factory. Finally, it is reasonable to assume that the production system satisfies the demand of goods from the market, that can be represented by a vector $\mathbf{b}= (b_i) \in \mathbb{R}^n$ (the vector of the *final demand*). The component b_i represents the quantity of the ith good absorbed by the market. The equilibrium is reached when the vector $\mathbf{x}= (x_i) \in \mathbb{R}^m$ of the total production equals the total demand, that is,

$$\mathrm{A}\mathbf{x} = \mathbf{b}, \qquad \text{where } \mathrm{A} = \mathrm{P} - \mathrm{C}. \tag{5.4}$$

For simplicity we will assume that ith factory produces only the ith good (see Figure 5.2). Consequently, $n = m$ and P = I. For the solution of this linear system see Exercise 5.19. ■

[1] On 1973 Wassily Leontief was awarded the Nobel prize in Economy for his studies. a linear production model for which

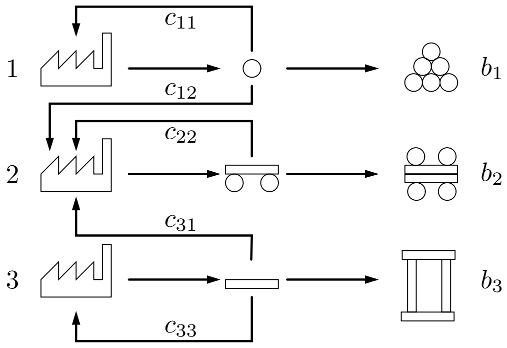


Figure 5.2. The interaction scheme between three factories and the market

Problem 5.4 (Capillary networks) Capillaries are tiny blood vessels, the smallest units of the blood circulatory system. They group together giving rise to networks called capillary beds featuring a variable number of elements, say from 10 to 100, depending upon the kind of organ and the specific biological tissue. The oxygenated blood reaches capillary beds from the arterioles, and from capillary beds it is released to the surrounding tissue passing through the membrane of red blood cells. Meanwhile, metabolic wastes are eliminated from the tissue by flowing into the capillary bed where it is gathered into small venules and eventually conveyed to the heart and from there to lungs. A capillary bed can be described by a network, similar to the hydraulic network considered in Problem 5.1; in this model, every capillary is assimilated to a pipeline whose endpoints are called nodes. In the schematic illustration of Figure 5.4, nodes are represented by empty little circles. From a functional viewpoint, the arteriole feeding the capillary bed can be regarded as a reservoir at uniform pressure (about 50 mmHg - note that one atmosphere corresponds to 760 mmHg). In our model we will assume that at the exiting nodes (those indicated by small black circles in Figure 5.4) the pressure features a constant value, that of the venous pressure, that we can normalize to zero. Blood flows from arterioles to the exiting nodes because of the pressure gap between one node and the following ones (those standing at a hierarchically lower level).

Still referring to Figure 5.4, we denote by $p_j,\ j = 1, ..., 15$ (measured in mmHg) the pressure at the jth node and by $Q_m,\ m = 1, ..., 31$ (measured in mm^3/s) the flow inside the mth capillary vessel. For any m, denoting by i and j the end-points of the mth capillary, we adopt the following constitutive relation

$$Q_m = \frac{1}{R_m L_m}(p_i - p_j), \tag{5.5}$$

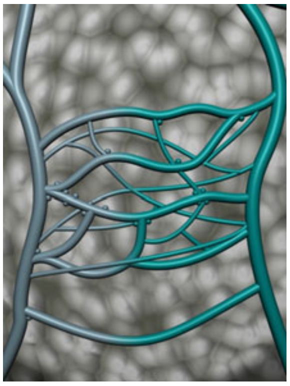
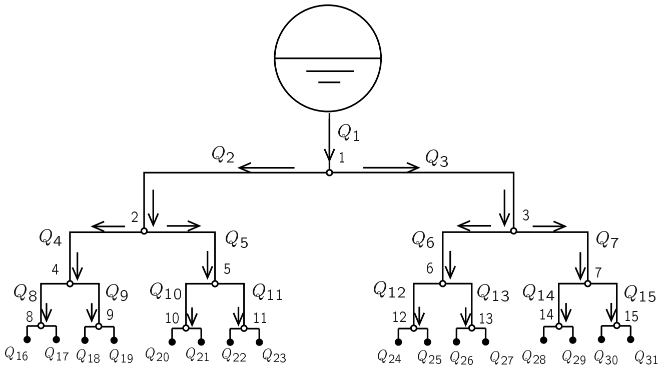


Figure 5.3. A capillary bed



Q_1
1
Q_2
Q_3
2
3
Q_4
Q_5
Q_6
Q_7
4
5
6
7
Q_8
Q_9
Q_{10}
Q_{11}
Q_{12}
Q_{13}
Q_{14}
Q_{15}
8
9
10
11
12
13
14
15
Q_{16} Q_{17} Q_{18} Q_{19} Q_{20} Q_{21} Q_{22} Q_{23}
Q_{24} Q_{25} Q_{26} Q_{27} Q_{28} Q_{29} Q_{30} Q_{31}

Figure 5.4. Schematization of a capillary bed

where R_m denotes the hydraulic resistance per unit length (in (mmHg s)/mm^4) and L_m the capillary length (in mm). Obviously, in considering the node number 1, we will take into account $p_0 = 50$; similarly, in considering the nodes from n. 8 to n. 15, we will set null pressure at outflow nodes (from n. 16 to n. 31). Finally, at any node of the network we will impose a balance equation between inflow and outflow, i.e.

$$\left(\sum_{m \text{ in}} Q_m\right) - \left(\sum_{m \text{ out}} Q_m\right) = 0.$$

In this way the following linear system is obtained

$$\mathrm{A}\mathbf{p} = \mathbf{b}, \tag{5.6}$$

where $\mathbf{p} = [p_1, p_2, \cdots, p_{15}]^T$ is the unknown vector of pressure at 15 nodes of the network, A is the matrix, while $\mathbf{b}$ is the array of known data.

For simplicity, let us suppose that all capillaries have the same hydraulic resistance $R_m = 1$ and that the length of the first capillary is $L_1 = 20$, while capillary length halves at each bifurcation (then $L_2 = L_3 = 10$, $L_4 = \ldots = L_7 = 5$ etc.). The following matrix is generated

$$A = \begin{bmatrix}
-\frac{1}{4} & \frac{1}{10} & \frac{1}{10} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{1}{10} & -\frac{1}{2} & 0 & \frac{1}{5} & \frac{1}{5} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{1}{10} & 0 & -\frac{1}{2} & 0 & 0 & \frac{1}{5} & \frac{1}{5} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & \frac{1}{5} & 0 & -1 & 0 & 0 & 0 & 0.4 & 0.4 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & \frac{1}{5} & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0.4 & 0.4 & 0 & 0 & 0 & 0 \\
0 & 0 & \frac{1}{5} & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0.4 & 0.4 & 0 & 0 \\
0 & 0 & \frac{1}{5} & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0.4 & 0.4 \\
0 & 0 & 0 & 0.4 & 0 & 0 & 0 & -2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0.4 & 0 & 0 & 0 & 0 & -2 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0.4 & 0 & 0 & 0 & 0 & -2 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0.4 & 0 & 0 & 0 & 0 & 0 & -2 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0.4 & 0 & 0 & 0 & 0 & 0 & -2 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0.4 & 0 & 0 & 0 & 0 & 0 & 0 & -2 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0.4 & 0 & 0 & 0 & 0 & 0 & 0 & -2 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0.4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -2
\end{bmatrix}$$

while $\mathbf{b} = [-5/2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]^T$.
The solution of this system will be considered in Example 5.7. ■

5.2 Linear system and complexity

The solution of system (5.1) exists iff A is nonsingular. In principle, it might be computed using the so-called *Cramer rule*

$$x_i = \frac{\det(\mathrm{A}_i)}{\det(\mathrm{A})}, \quad i = 1, \ldots, n,$$

where A_i is the matrix obtained from A by replacing the ith column by $\mathbf{b}$ and det(A) denotes the determinant of A. If the $n+1$ determinants are computed by the Laplace expansion (see Exercise 5.1), a total number of approximately $3(n+1)!$ operations is required. As usual, by operation we mean a sum, a subtraction, a product or a division. For instance, a computer capable of carrying out 10^9 *flops* (i.e. 1 Giga *flops*), would require about 17 hours to solve a system of dimension $n = 15$, 4860 years if $n = 20$ and 10^{143} years if $n = 100$. See Table 5.1. Note that 10^9 flops is the characteristic speed of a current PC (e.g. with a processor Intel® Core™2 Duo, 2.53 GHz) whereas the Tianhe-2 (MilkyWay-2) Cluster of the National University of Defense Technology in China, 1st of the top500 supercomputer list as of November 2013, features a rate of 33 Peta-flops (i.e. $33 \cdot 10^{15}$ flops).

Table 5.1. Time required to solve a linear system of dimension n by the Cramer rule. "o.r." stands for "out of reach"

	Flops				
n	10^9 (Giga)	10^{10}	10^{11}	10^{12} (Tera)	10^{15} (Peta)
10	10^{-1}sec	10^{-2}sec	10^{-3}sec	10^{-4}sec	negligible
15	17 hours	1.74 hours	10.46 min	1 min	$0.6\ 10^{-1}$ sec
20	4860 years	486 years	48.6 years	4.86 years	1.7 day
25	o.r.	o.r.	o.r.	o.r.	38365 years

The computational cost can be drastically reduced to the order of about $n^{3.8}$ operations if the $n+1$ determinants are computed by the algorithm quoted in Example 1.3. Yet, this cost is still too high for large values of n, which often arise in practical applications.

Two alternative approaches will be pursued: they are called *direct methods* if they yield the solution of the system in a finite number of steps, *iterative methods* if they require (in principle) an infinite number of steps. Iterative methods will be addressed in Section 5.9. We warn the reader that the choice between direct and iterative methods may depend on several factors: primarily, the predicted theoretical efficiency of the scheme, but also the particular type of matrix, the memory storage requirements and, finally, the computer architecture (see, Section 5.13 for more details).

Finally, we note that a system with full matrix cannot be solved by less than n^2 operations. Indeed, if the equations are fully coupled, we should expect that every one of the n^2 matrix coefficients would be involved in an algebraic operation at least once.

Even though most of the methods that we will present in this Section are valid for compex matrices too, for simplicity we will limit our analysis to real matrices. In any case, we note that MATLAB and Octave programs for solving linear systems work on both real and complex variables, with no need to modify the calling instructions.

We will explicitly refer to complex matrices only when the assumptions that we have to make on real matrices have to be replaced by specific conditions in the complex field. This happens, for instance, when we have to define positive definite matrices, or when we need to specify the conditions that underly the Cholesky factorization of a matrix.

5.3 The LU factorization method

Let $\mathrm{A}\in \mathbb{R}^{n\times n}$. Assume that there exist two suitable matrices L and U, lower triangular and upper triangular, respectively, such that

$$\boxed{\mathrm{A} = \mathrm{LU}} \tag{5.7}$$

We call (5.7) an LU-*factorization* (or decomposition) of A. If A is non-singular, so are both L and U, and thus their diagonal elements are non-null (as observed in Section 1.4).

In such a case, solving $\mathrm{A}\mathbf{x} = \mathbf{b}$ leads to the solution of the two triangular systems

$$\boxed{\mathrm{L}\mathbf{y} = \mathbf{b}, \qquad \mathrm{U}\mathbf{x} = \mathbf{y}} \tag{5.8}$$

Both systems are easy to solve. Indeed, L being lower triangular, the first row of the system $\mathrm{L}\mathbf{y} = \mathbf{b}$ takes the form:

$$l_{11}y_1 = b_1,$$

which provides the value of y_1 since $l_{11} \neq 0$. By substituting this value of y_1 in the subsequent $n-1$ equations we obtain a new system whose unknowns are $y_2, \ldots, y_n$, on which we can proceed in a similar manner. Proceeding forward, equation by equation, we can compute all unknowns with the following *forward substitutions algorithm*:

$$\boxed{\begin{aligned} y_1 &= \frac{1}{l_{11}} b_1, \\ y_i &= \frac{1}{l_{ii}} \left(b_i - \sum_{j=1}^{i-1} l_{ij} y_j \right), \ i = 2, \ldots, n \end{aligned}} \tag{5.9}$$

Let us count the number of operations required by (5.9). Since $i-1$ sums, $i-1$ products and 1 division are needed to compute the unknown y_i, the total number of operations required is

$$\sum_{i=1}^{n} 1 + 2\sum_{i=1}^{n}(i-1) = 2\sum_{i=1}^{n} i - n = n^2.$$

The system $\mathrm{U}\mathbf{x} = \mathbf{y}$ can be solved by proceeding in a similar manner. This time, the first unknown to be computed is x_n, then, by proceeding backward, we can compute the remaining unknowns x_i, for $i = n-1$ to $i = 1$:

$$\boxed{\begin{aligned} x_n &= \frac{1}{u_{nn}} y_n, \\ x_i &= \frac{1}{u_{ii}} \left(y_i - \sum_{j=i+1}^{n} u_{ij} x_j \right), \ i = n-1, \ldots, 1 \end{aligned}} \tag{5.10}$$

This is called *backward substitutions algorithm* and requires n^2 operations too. At this stage we need an algorithm that allows an effective computation of the factors L and U of the matrix A. We illustrate a general procedure starting from a couple of examples.

Example 5.1 Let us write the relation (5.7) for a generic matrix $\mathrm{A} \in \mathbb{R}^{2\times 2}$

$$\begin{bmatrix} l_{11} & 0 \\ l_{21} & l_{22} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} \\ 0 & u_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}.$$

The 6 unknown elements of L and U must satisfy the following (nonlinear) equations:

$$\begin{array}{ll} (e_1) \;\; l_{11}u_{11} = a_{11}, & (e_2) \;\; l_{11}u_{12} = a_{12}, \\ (e_3) \;\; l_{21}u_{11} = a_{21}, & (e_4) \;\; l_{21}u_{12} + l_{22}u_{22} = a_{22}. \end{array} \tag{5.11}$$

System (5.11) is *underdetermined* as it features less equations than unknowns. We can complete it by assigning *arbitrarily* the diagonal elements of L, for instance setting $l_{11} = 1$ and $l_{22} = 1$. Now system (5.11) can be solved by proceeding as follows: we determine the elements u_{11} and u_{12} of the first row of U using (e_1) and (e_2). If u_{11} is non-null then from (e_3) we deduce l_{21} (that is the first column of L, since l_{11} is already available). Now we can obtain from (e_4) the only nonzero element u_{22} of the second row of U. ■

Example 5.2 Let us repeat the same computations in the case of a 3×3 matrix. For the 12 unknown coefficients of L and U we have the following 9 equations:

$$\begin{array}{lll} (e_1)\; l_{11}u_{11} = a_{11}, & (e_2)\; l_{11}u_{12} = a_{12}, & (e_3)\; l_{11}u_{13} = a_{13}, \\ (e_4)\; l_{21}u_{11} = a_{21}, & (e_5)\; l_{21}u_{12} + l_{22}u_{22} = a_{22}, & (e_6)\; l_{21}u_{13} + l_{22}u_{23} = a_{23}, \\ (e_7)\; l_{31}u_{11} = a_{31}, & (e_8)\; l_{31}u_{12} + l_{32}u_{22} = a_{32}, & (e_9)\; l_{31}u_{13} + l_{32}u_{23} + l_{33}u_{33} = a_{33}. \end{array}$$

Let us complete this system by setting $l_{ii} = 1$ for $i = 1, 2, 3$. Now, the coefficients of the first row of U can be obtained by using (e_1), (e_2) and (e_3). Next, using (e_4) and (e_7), we can determine the coefficients l_{21} and l_{31} of the first column of L. Using (e_5) and (e_6) we can now compute the coefficients u_{22} and u_{23} of the second row of U. Then, using (e_8), we obtain the coefficient l_{32} of the second column of L. Finally, the last row of U (which consists of the only element u_{33}) can be determined by solving (e_9). ■

On a matrix A$\in \mathbb{R}^{n\times n}$ of arbitrary dimension n we can proceed as follows:

1. the elements of L and U satisfy the system of nonlinear equations

$$\sum_{r=1}^{\min(i,j)} l_{ir}u_{rj} = a_{ij},\; i, j = 1, \ldots, n; \tag{5.12}$$

2. system (5.12) is underdetermined; indeed, there are n^2 equations and $n^2 + n$ unknowns. Consequently, the LU factorization cannot be unique; more precisely, infinite pairs of matrices L and U satisfying (5.12) can exist;
3. by forcing the n diagonal elements of L to be equal to 1, (5.12) turns into a determined system which can be solved by the following *Gauss algorithm*, set $\mathrm{A}^{(1)} = \mathrm{A}$ i.e. $a_{ij}^{(1)} = a_{ij}$ for $i, j = 1, \ldots, n$;

```
for $k = 1, \ldots, n-1$
    for $i = k+1, \ldots, n$
        $l_{ik} = \dfrac{a_{ik}^{(k)}}{a_{kk}^{(k)}}$,
        for $j = k+1, \ldots, n$
            $a_{ij}^{(k+1)} = a_{ij}^{(k)} - l_{ik} a_{kj}^{(k)}$
```
(5.13)

The elements $a_{kk}^{(k)}$ must all be different from zero and are called *pivot elements*. For every $k = 1, \ldots, n-1$ the matrix $\mathrm{A}^{(k+1)} = (a_{ij}^{(k+1)})$ has $n-k$ rows and columns.

At the end of this procedure the elements of the upper triangular matrix U are given by $u_{ij} = a_{ij}^{(i)}$ for $i = 1, \ldots, n$ and $j = i, \ldots, n$, whereas those of L are given by the coefficients l_{ij} generated by this algorithm. In (5.13) there is no computation of the diagonal elements of L, as we already know that their value is equal to 1.

This LU factorization is also called *Gauss LU factorization*; from now on we will simply call it LU factorization. Determining the elements of the factors L and U requires about $2n^3/3$ operations (see Exercise 5.4).

Remark 5.1 Storing all the matrices $\mathrm{A}^{(k)}$ in the algorithm (5.13) is not necessary; actually we can overlap the $(n-k) \times (n-k)$ elements of $\mathrm{A}^{(k+1)}$ on the corresponding last $(n-k) \times (n-k)$ elements of the original matrix A. Moreover, since at step k, the subdiagonal elements of the kth column don't have any effect on the final U, they can be replaced by the entries of the kth column of L (the so-called *multipliers*), as done in Program 5.1. Then, at step k of the process the elements stored at location of the original entries of A are

$$\left[\begin{array}{cccccc}
a_{11}^{(1)} & a_{12}^{(1)} & \ldots & \ldots & \ldots & a_{1n}^{(1)} \\
l_{21} & a_{22}^{(2)} & & & & a_{2n}^{(2)} \\
\vdots & \ddots & \ddots & & & \vdots \\
l_{k1} & \ldots & l_{k,k-1} & \boxed{a_{kk}^{(k)}} & \ldots & a_{kn}^{(k)} \\
\vdots & & \vdots & \vdots & & \vdots \\
l_{n1} & \ldots & l_{n,k-1} & a_{nk}^{(k)} & \ldots & a_{nn}^{(k)}
\end{array}\right],$$

where the boxed submatrix is $\mathrm{A}^{(k)}$.

More precisely, this algorithm can be implemented by storing a unique matrix, which is initialized equal to A and then modified at each step $k \geq 2$ by overwriting the new entries $a_{ij}^{(k)}$, for $i, j \geq k+1$, as well as the multipliers l_{ik}, for $i \geq k+1$. Note that it is not indispensable to store diagonal elements l_{ii}; in fact, it is understood that they are all equal to 1. ■

Remark 5.2 (Gauss Elimination Method (GEM)) Gauss algorithm (5.13) can be applied also to the right hand side of the linear system $\mathrm{A}\mathbf{x} = \mathbf{b}$ as follows:

set $\mathrm{A}^{(1)} = \mathrm{A}$ i.e. $a_{ij}^{(1)} = a_{ij}$ for $i, j = 1, \dots, n$; set $\mathbf{b}^{(1)} = \mathbf{b}$ i.e. $b_i^{(1)} = b_i$ for $i = 1, \dots, n$;

```
for $k = 1, \dots, n-1$
    for $i = k+1, \dots, n$
        $l_{ik} = \dfrac{a_{ik}^{(k)}}{a_{kk}^{(k)}},$
        for $j = k+1, \dots, n$
            $a_{ij}^{(k+1)} = a_{ij}^{(k)} - l_{ik} a_{kj}^{(k)}$
        $b_i^{(k+1)} = b_i^{(k)} - l_{ik} b_k^{(k)}$
```
(5.14)

This algorithm is called *Gauss Elimination Methods (GEM)*. At the end of the process, the upper triangular system $\mathbf{A}^{(n)}\mathbf{x} = \mathbf{b}^{(n)}$, which is equivalent to the original one, can be solved by the backward substitution algorithm (5.10). ■

Example 5.3 (Spectrometry) For the Problem 5.2 we consider a gas mixture that, after a spectroscopic inspection, presents the following seven most relevant peaks: $h_1 = 17.1$, $h_2 = 65.1$, $h_3 = 186.0$, $h_4 = 82.7$, $h_5 = 84.2$, $h_6 = 63.7$ and $h_7 = 119.7$. We want to compare the measured total pressure, equal to 38.78 μm of Hg (which accounts also for those components that we might have neglected in our simplified model) with that obtained using relations (5.3) with $n = 7$, where the sensitivity coefficients are given in Table 5.2 (taken from [CLW69, p. 331]). The partial pressures can be computed solving the system (5.3) for $n = 7$ using the LU factorization. We obtain

```
partpress=
    0.6525
    2.2038
    0.3348
    6.4344
    2.9975
    0.5505
   25.6317
```

Using these values we compute an approximate total pressure (given by `sum(partpress)`) of the gas mixture which differs from the measured value by 0.0252 μm of Hg. ■

Example 5.4 Consider the Vandermonde matrix

$$\mathrm{A} = (a_{ij}) \text{ with } a_{ij} = x_i^{n-j}, \; i, j = 1, \dots, n, \tag{5.15}$$

where the x_i are n distinct abscissae. It can be constructed using the MATLAB command `vander`. In Table 5.3 we report the time required to compute the LU factorization of A (which behaves like $2n^3/3$, see Figure 5.5) on different

`vander`

Table 5.2. The sensitivity coefficients for a gas mixture

	Components and indices						
Peak	Hydrogen	Methane	Etilene	Ethane	Propylene	Propane	n-Pentane
index	1	2	3	4	5	6	7
1	16.87	0.1650	0.2019	0.3170	0.2340	0.1820	0.1100
2	0.0	27.70	0.8620	0.0620	0.0730	0.1310	0.1200
3	0.0	0.0	22.35	13.05	4.420	6.001	3.043
4	0.0	0.0	0.0	11.28	0.0	1.110	0.3710
5	0.0	0.0	0.0	0.0	9.850	1.1684	2.108
6	0.0	0.0	0.0	0.0	0.2990	15.98	2.107
7	0.0	0.0	0.0	0.0	0.0	0.0	4.670

Table 5.3. Time required to solve a full linear system of dimension n by MEG. "o.r." stands for "out of reach"

		Flops	
n	10^9 (Giga)	10^{12} (Tera)	10^{15} (Peta)
10^2	$7 \cdot 10^{-4}$sec	negligible	negligible
10^4	11 min	0.7 sec	$7 \cdot 10^{-4}$sec
10^6	21 years	7.7 months	11 min
10^8	o.r.	o.r.	21 years

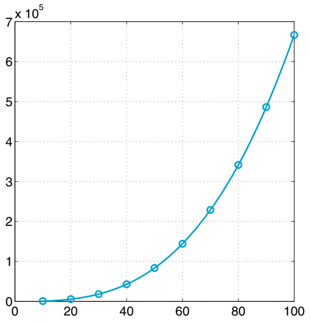


Figure 5.5. The number of floating-point operations necessary to generate the LU factorization of the Vandermonde matrix, as a function of the matrix dimension n. This function is a cubic polynomial obtained by approximating in the least-squares sense the values (represented by circles for $n = 10, 20, \ldots, 100$)

computers featuring 1 GigaFlops, 1 TeraFlops and 1PetaFlops performance, respectively. In Figure 5.5 we plot the number of floating-point operations necessary to generate the LU factorization of the Vandermonde matrix, as a function of the matrix dimension n. These values were provided by the command `flops` that was present in former versions of MATLAB. ■

The LU factorization is the basis of several MATLAB commands:

- `[L,U]=lu(A)` whose mode of use will be discussed in Section 5.4; lu
- `inv` that allows the computation of the inverse of a matrix; inv
- `\` by which it is possible to solve a linear system with matrix `A` and right hand side `b` by simply writing `A\b` (see Section 5.8). \

Remark 5.3 (Computing a determinant) By means of the LU factorization one can compute the determinant of A with a computational cost of $\mathcal{O}(n^3)$ operations, noting that (see Sect.1.4)

$$\det(\mathrm{A}) = \det(\mathrm{L})\ \det(\mathrm{U}) = \prod_{k=1}^{n} u_{kk}.$$

As a matter of fact, this procedure is also at the basis of the MATLAB command `det`. ■

In Program 5.1 we implement the algorithm (5.13). The factor `L` is stored in the (strictly) lower triangular part of `A` and `U` in the upper triangular part of `A` (for the sake of storage saving). After the program execution, the two factors can be recovered by simply writing: `L = eye(n) + tril(A,-1)` and `U = triu(A)`, where `n` is the size of `A`.

Program 5.1. lugauss: Gauss LU factorization

```
function A=lugauss(A)
%LUGAUSS LU factorization without pivoting.
% A = LUGAUSS(A) stores an upper triangular matrix in
% the upper triangular part of A and a lower triangular
% matrix in the strictly lower part of A (the diagonal
% elements of L are 1).
[n,m]=size(A);
if n ~= m; error('A is not a square matrix'); else
 for k = 1:n-1
  for i = k+1:n
   A(i,k) = A(i,k)/A(k,k);
   if A(k,k) == 0, error('Null diagonal element'); end
   j = [k+1:n]; A(i,j) = A(i,j) - A(i,k)*A(k,j);
  end
 end
end
```

Example 5.5 Let us compute the solution of the system encountered in Problem 5.1 by using the LU factorization, then applying the backward and forward substitution algorithms. We need to compute the matrix `A` and the right-hand side `b` and execute the following instructions:

```
A=lugauss(A);   y(1)=b(1);
for i=2:4; y=[y; b(i)-A(i,1:i-1)*y(1:i-1)]; end
x(4)=y(4)/A(4,4);
for i=3:-1:1
x(i)=(y(i)-A(i,i+1:4)*x(i+1:4)')/A(i,i); end
p=x
```

The result is $\mathbf{p} = (8.1172, 5.9893, 5.9893, 5.7779)^T$. ■

Example 5.6 Suppose that we solve $\mathrm{A}\mathbf{x} = \mathbf{b}$ with

$$\mathrm{A} = \begin{bmatrix} 1 & 1-\varepsilon & 3 \\ 2 & 2 & 2 \\ 3 & 6 & 4 \end{bmatrix}, \mathbf{b} = \begin{bmatrix} 5-\varepsilon \\ 6 \\ 13 \end{bmatrix}, \varepsilon \in \mathbb{R}, \tag{5.16}$$

whose solution is $\mathbf{x} = (1, 1, 1)^T$ (independently of the value of ε).
Let us set $\varepsilon = 1$. The LU factorization of A obtained by the Program 5.1 yields

$$\mathrm{L} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & 3 & 1 \end{bmatrix}, \mathrm{U} = \begin{bmatrix} 1 & 0 & 3 \\ 0 & 2 & -4 \\ 0 & 0 & 7 \end{bmatrix}.$$

If we set $\varepsilon = 0$, despite the fact that A is non singular, the LU factorization cannot be carried out since the algorithm (5.13) would involve divisions by 0. ■

The previous example shows that, unfortunately, the LU factorization A=LU does not necessarily exist for every nonsingular matrix A. In this respect, the following result can be proven:

Proposition 5.1 *For a given matrix* $\mathrm{A} \in \mathbb{R}^{n\times n}$*, its LU factorization exists and is unique iff the principal submatrices* A_i *of A of order* $i = 1, \ldots, n-1$ *(that is those obtained by restricting A to its first i rows and columns) are nonsingular. (This result holds also for any* $\mathrm{A} \in \mathbb{C}^{n\times n}$ *[Zha99, Sect. 3.2].)*

Going back to Example 5.6, we can notice that when $\varepsilon = 0$ the second principal submatrix A_2 of the matrix A is singular.

We can identify special classes of matrices for which the hypotheses of Proposition 5.1 are fulfilled. In particular, we mention:

1. strictly diagonally dominant matrices.
   A matrix is *diagonally dominant by row* if

$$|a_{ii}| \geq \sum_{\substack{j=1\\ j\neq i}}^{n} |a_{ij}|, \quad i = 1, \ldots, n,$$

   by column if

$$|a_{ii}| \geq \sum_{\substack{j=1\\ j\neq i}}^{n} |a_{ji}|, \quad i = 1, \ldots, n.$$

   When in the previous inequalities we can replace $\geq$ by $>$ we say that matrix A is *strictly diagonally dominant* (by row or by column,

respectively). This definition holds also for any matrix A$\in \mathbb{C}^{n\times n}$ (see [GI04]);
2. real symmetric and positive definite matrices. A matrix A $\in \mathbb{R}^{n\times n}$ is *positive definite* if

$$\forall \mathbf{x} \in \mathbb{R}^n \text{ with } \mathbf{x} \neq \mathbf{0}, \qquad \mathbf{x}^T \mathrm{A}\mathbf{x} > 0$$

and *semi positive definite* if

$$\forall \mathbf{x} \in \mathbb{R}^n, \qquad \mathbf{x}^T \mathrm{A}\mathbf{x} \geq 0;$$

3. complex definite positive matrices $A \in \mathbb{C}^{n\times n}$, that is

$$\forall \mathbf{x} \in \mathbb{C}^n \text{ with } \mathbf{x} \neq \mathbf{0}, \qquad \mathbf{x}^H \mathrm{A}\mathbf{x} > 0;$$

note that these matrices are necessarily hermitian matrices (see [Zha99, Sect. 3.2]).

If A$\in \mathbb{R}^{n\times n}$ is symmetric and positive definite, it is moreover possible to construct a special factorization:

$$\boxed{\mathrm{A} = \mathrm{R}^T\mathrm{R}} \tag{5.17}$$

where R is an upper triangular matrix with positive diagonal elements. This is the so-called *Cholesky factorization* and requires about $n^3/3$ operations (half of those required by the LU factorization). Further, let us note that, due to the symmetry, only the upper part of A is stored, and R can be stored in the same area.

The elements of R can be computed by the following algorithm: we set $r_{11} = \sqrt{a_{11}}$ and, for $j = 2, \ldots, n$, we set

$$\boxed{\begin{aligned} r_{ij} &= \frac{1}{r_{ii}}\left(a_{ij} - \sum_{k=1}^{i-1} r_{ki}r_{kj}\right), \; i = 1, \ldots, j-1 \\ r_{jj} &= \sqrt{a_{jj} - \sum_{k=1}^{j-1} r_{kj}^2} \end{aligned}} \tag{5.18}$$

Cholesky factorization is available in MATLAB by setting `R=chol(A)`. If we consider a complex positive definite matrix A$\in \mathbb{C}^{n\times n}$, formula (5.17) becomes A=R^{H}R, R^H being the conjugate transpose matrix of R.

`chol`

Example 5.7 (Capillary networks) Consider problem 5.4 and change the sign of the associated system (5.6). It turns out that $-$A is symmetric positive definite, hence the system $-\mathrm{A}\mathbf{p} = -\mathbf{b}$ can be solved by Cholesky factorization. The corresponding solution is given by the following vector

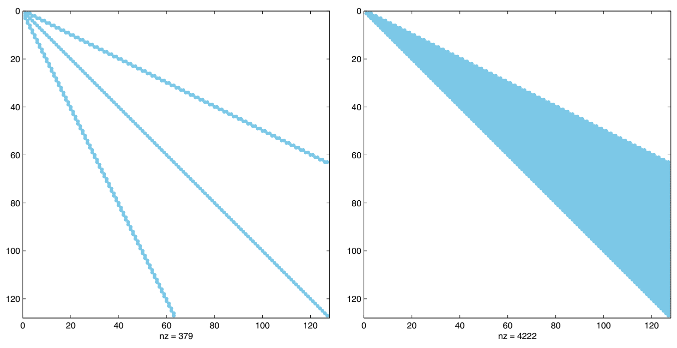


Figure 5.6. Pattern of matrices A and R of Example 5.7

$$\mathbf{p} = [12.46, 3.07, 3.07, .73, .73, .73, .15, .15, .15, .15, .15, .15, .15, .15, .15]^T.$$

Consequently, owing to relations (5.5), the following flow-rates are found:

$$\begin{array}{lcl} Q_1 & = & 1.88 \\ Q_{2,3} & = & 0.94 \\ Q_{4,\cdots,7} & = & 0.47 \\ Q_{8,\cdots,15} & = & 0.23 \\ Q_{16,\cdots,31} & = & 0.12. \end{array}$$

Matrix A features a special banded structure, see, e.g., Figure 5.6 for an instance corresponding to a capillary bed with 8 bifurcation levels. Colored dots corresponds to non-null entries of A. On each row, there are at most 3 non-null entries. Besides, A is sparse (see at the end of this Example for the definition of sparse matrices), as it features only 379 non-null entries over a total of $(127)^2 = 16129$ elements. The Cholesky factorization generates fill-in inside bands (see Sect. 5.4.1), as we can see from Figure 5.6 (right), where the sparsity pattern of the upper triangular Cholesky factor R is shown. Reduction of fill-in is possibile provided suitable reordering algorithms are used on the given matrix A. An example is given in Figure 5.7, where at left we display the reordered matrix A (corresponding to the original matrix of Figure 5.6, left) and at right the corresponding upper Cholesky factor R. For a discussion about ordering techniques we refer the interested reader to [QSS07, Sect. 3.9]. ■

A square matrix of size n is said *sparse* if the number of its nonzero entries is of order n (therefore asymptotically less than the total number n^2 of the coefficients of A). The *pattern* of a sparse matrix A is the 2D graphical representation of the positions of its nonnull entries. For instance, the pattern of matrices A and R of Example 5.7 is shown in Fig. 5.6; it is plotted by invoking the commands `spy(A)` and `spy(R)`, respectively.

spy

A matrix $A \in \mathbb{R}^{m\times n}$ (or in $\mathbb{C}^{m\times n}$) has *lower band* p if $a_{ij} = 0$ when $i > j + p$ and *upper band* q if $a_{ij} = 0$ when $j > i + q$. The maximum between p and q is called the *bandwidth* of the matrix.

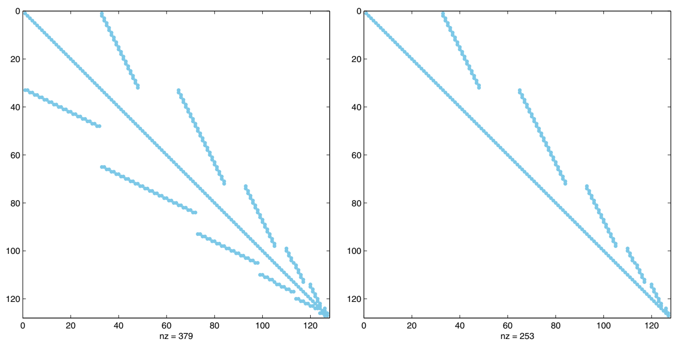


Figure 5.7. Pattern of matrices A and R of Example 5.7 after reordering

If A is a banded or a sparse large size matrix, only its non-zero entries need to be stored. This can be conveniently done by means of the MATLAB commands `sparse` or `spdiags`. For instance, the command

sparse
spdiags

```
A=sparse(n,m)
```

initializes to zero each entry of a *sparse-array* (a specific MATLAB variable) of n rows and m columns.
The square matrix A such that

$$\begin{array}{ll} a_{ii}=1 & \text{for } i=1,\ldots,n,\\ a_{1j}=1 & \text{for } j=1,\ldots,n,\\ a_{i1}=1 & \text{for } i=1,\ldots,n,\\ a_{ij}=0 & \text{otherwise} \end{array} \tag{5.19}$$

with $n=25$ can be defined as follows:

```
n=25;   e=ones(n,1);
A=spdiags(e,0,n,n);
A(1,:)=e'; A(:,1)=e;
```

The command `spdiags` initializes a matrix with `n` rows and columns according to the *sparse-array* format by positioning the column vector `e` on the main diagonal (of index 0), then the following instructions (that are valid for all kind of MATLAB arrays) update the first row and the first column, respectively, of A.

When a system is solved by invoking the command \, MATLAB is able to recognize the type of matrix and, in particular, whether it has been stored as a sparse-array; consequently, MATLAB sorts out the most appropriate solution algorithm as we will see in Sect. 5.8.

See Exercises 5.1-5.5.

5.4 The pivoting technique

We are going to introduce a special technique that allows us to achieve the LU factorization for every nonsingular matrix, even when the assumptions of Proposition 5.1 are not fulfilled.

Let us go back for a while to the case described in Example 5.6 and take $\varepsilon = 0$. Setting $\mathrm{A}^{(1)} = \mathrm{A}$ after carrying out the first step ($k = 1$) of the procedure, the new entries of A are

$$\begin{bmatrix}\begin{array}{|c|cc|}\hline 1 & 1 & 3\\ \hline 2 & \mathbf{0} & -4\\ 3 & 3 & -5\\ \hline\end{array}\end{bmatrix}. \tag{5.20}$$

Since the *pivot* a_{22} is equal to zero, this procedure cannot be continued further on. However, should we interchange the second and third rows beforehand, we would obtain the matrix

$$\begin{bmatrix}\begin{array}{|c|cc|}\hline 1 & 1 & 3\\ \hline 3 & \mathbf{3} & -5\\ 2 & 0 & -4\\ \hline\end{array}\end{bmatrix}$$

and thus the factorization could be accomplished without involving a division by 0.

We can state that a suitable *row permutation* of the original matrix A would make the entire factorization procedure feasible even if the hypotheses of Proposition 5.1 are not verified, under the sole condition that $\det(\mathrm{A}) \neq 0$. The decision on which row to permute can be made at every step k at which a null diagonal element $a_{kk}^{(k)}$ is generated.

Let us return to matrix (5.20), whose (2,2) coefficient is null. As the element (3,2) is not null, we interchange the third and second row of this matrix. By executing the second step of the factorization procedure we finally find the same matrix that we would have generated by an *a priori* permutation of the same two rows of A.

Since a row permutation entails changing the *pivot element*, this technique is given the name of *pivoting by row*. The factorization generated in this way returns the original matrix up to a row permutation. Precisely we obtain

$$\boxed{\mathrm{PA} = \mathrm{LU}} \tag{5.21}$$

where P is a suitable *permutation matrix* that is initially set equal to the identity matrix, then whenever in the course of the procedure two rows of A are permuted, the same permutation must be performed on the corresponding rows of P. At last, because of (5.21), we should sequentially solve the following triangular systems

$$\mathrm{L}\mathbf{y} = \mathrm{P}\mathbf{b}, \qquad \mathrm{U}\mathbf{x} = \mathbf{y}. \tag{5.22}$$

From the second equation of (5.13) we see that not only null pivot elements $a_{kk}^{(k)}$ are troublesome, but so are those which are very small. Indeed, should $a_{kk}^{(k)}$ be near zero, possible roundoff errors affecting the coefficients $a_{kj}^{(k)}$ will be severely amplified.

Example 5.8 Consider the nonsingular matrix

$$\mathrm{A} = \begin{bmatrix} 1 & 1 + 0.5 \cdot 10^{-15} & 3 \\ 2 & 2 & 20 \\ 3 & 6 & 4 \end{bmatrix}.$$

Although during the factorization procedure by Program 5.1 no null pivot element is generated yet, the factors L and U turn out to be quite inaccurate, as one can realize by computing the residual matrix A − LU (which should be the null matrix if all operations were carried out in exact arithmetic):

$$\mathrm{A} - \mathrm{LU} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 4 \end{bmatrix}.$$

■

It is therefore recommended to carry out the pivoting at every step of the factorization procedure, by searching among all virtual pivot elements $a_{ik}^{(k)}$ with $i = k, \ldots, n$, the one with maximum modulus (see Fig. 5.8, left). The algorithm (5.13) with pivoting by row carried out at each step takes the following form: set $\mathrm{A}^{(1)} = \mathrm{A}$ and P=I, then:

```
for $k = 1, \ldots, n-1$,
    find $\bar{r}$ such that $|a_{\bar{r}k}^{(k)}| = \max_{r=k,\ldots,n} |a_{rk}^{(k)}|$,
    exchange row $k$ with row $\bar{r}$
    in both A and P,
    for $i = k+1, \ldots, n$
        $l_{ik} = \frac{a_{ik}^{(k)}}{a_{kk}^{(k)}}$,
        for $j = k+1, \ldots, n$
            $a_{ij}^{(k+1)} = a_{ij}^{(k)} - l_{ik} a_{kj}^{(k)}$
```
(5.23)

As already noticed for algorithm (5.13) (the one without permutations), a single matrix is sufficient to store either the entries $(a_{ij}^{(k)})$ and the multipliers (l_{ik}). Consequently, for any k, the same permutation carried out on both A and P acts on the multipliers, too.

The MATLAB program `lu` previously mentioned computes the LU factorization with pivoting by row. Its complete syntax is indeed

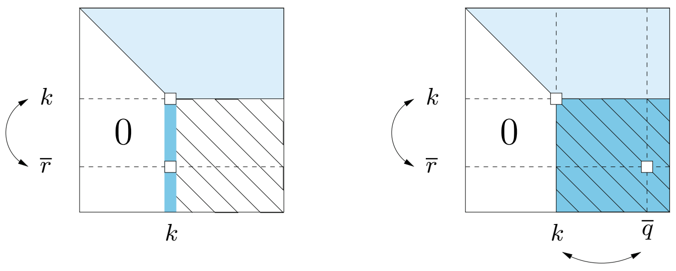


Figure 5.8. Partial pivoting by rows *(at left)* and total pivoting *(at right)*. The submatrix where to seek the *pivot* at step k is coloured by a dark blue

[L,U,P]=lu(A), P being the permutation matrix. When called in the shorthand mode [L,U]=lu(A), the matrix L is equal to P*M, where M is lower triangular and P is the permutation matrix generated by the pivoting by row. The program lu automatically operates pivoting by row. In particular, when A has a sparse storage organization (see Sections 5.6 and 5.8), permutation by rows is performed only when a null (or exceedingly small) pivot element is encountered.

Total pivoting consists of searching the *pivot* within the whole $\mathrm{A}^{(k)}$ submatrix made by the elements $a_{ij}^{(k)}$, $i, j = k, \ldots, n$ (see Fig. 5.8, right). It involves both the rows and the columns of the matrix and yields two permutation matrices, P on the rows and Q on the columns, such that

$$\boxed{\mathrm{PAQ} = \mathrm{LU}} \tag{5.24}$$

As

$$\mathrm{A}\mathbf{x} = \mathbf{b} \Leftrightarrow \underbrace{\mathrm{PAQ}}_{\mathrm{LU}}\,\underbrace{\mathrm{Q}^{-1}\mathbf{x}}_{\mathbf{x}^*} = \mathrm{P}\mathbf{b},$$

the solution of system $\mathrm{A}\mathbf{x} = \mathbf{b}$ is then obtained by solving two triangular systems and finally operating a permuation on the vector components as follows

$$\boxed{\mathrm{L}\mathbf{y} = \mathrm{P}\mathbf{b} \qquad \mathrm{U}\mathbf{x}^* = \mathbf{y} \qquad \mathbf{x} = \mathrm{Q}\mathbf{x}^*} \tag{5.25}$$

The MATLAB command [L,U,P,Q]=lu(A) implements the total pivoting on an input matrix featuring a sparse-array format (hence generated by the sparse or spdiags commands).

Total pivoting is computationally more expensive than partial pivoting as many more comparisons need to be carried out at every step of the factorization. However it can save storage and enhance the algorithm stability as we will see in the next sections.

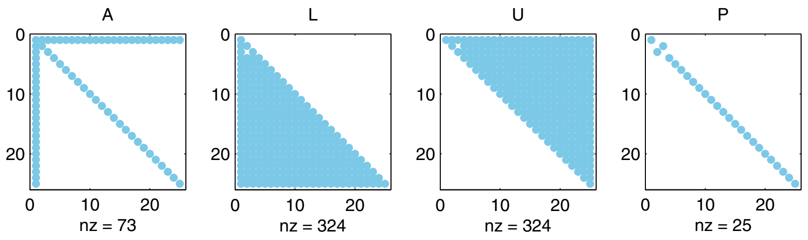


Figure 5.9. *Fill-in* of matrix A defined in (5.19)

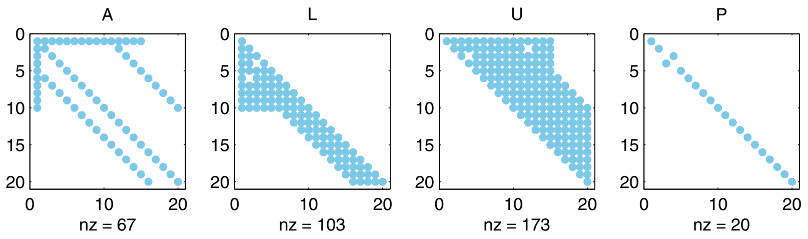


Figure 5.10. *Fill-in* for a matrix A whose profile is indicated in the first figure on the left

5.4.1 The *fill-in* of a matrix

In general, the LU factorization process does not preserve the sparsity pattern of the original matrix A, as it may generate non-null elements in L and U in some positions (i, j) where the corresponding original elements a_{ij} were null. This phenomenon is called *fill-in* and depends on both the original pattern of A and the values of its entries.

Examples of fill-in were already encountered in Figure 5.6 (referring to Example 5.7), while Figure 5.9 refers to the fill-in for the case of matrix (5.19). Another example is shown in Figure 5.10: in this case the non-null elements sitting on the first row and first column of A induce a complete fill-in of the corresponding columns of U and rows of L, respectively. Moreover, the non-null elements lying on the upper and lower diagonals of A yield a fill-in of the upper diagonals of U and the lower diagonals of L included between the main diagonal and those non-null of A.

The fill-in can be avoided by operating a suitable reordering (through row and column pemutations) of the matrix A before operating its factorization. In many cases, however, the sole total pivoting allows the achievement of the same result. An example is illustrated in Figure 5.11 which refers to the case of matrix A defined in (5.19). We can appreciate that no fill-in is induced on the factors L and U, at the cost however of

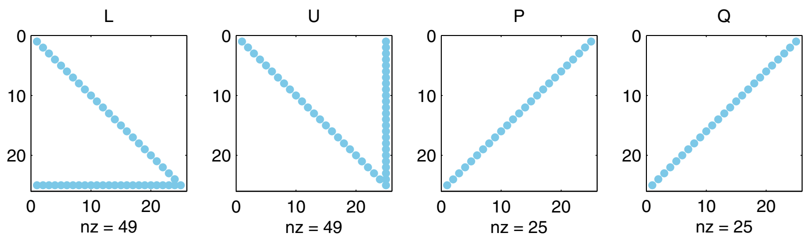


Figure 5.11. Matrices L, U, P and Q of the total pivoting factorization of A defined in (5.19)

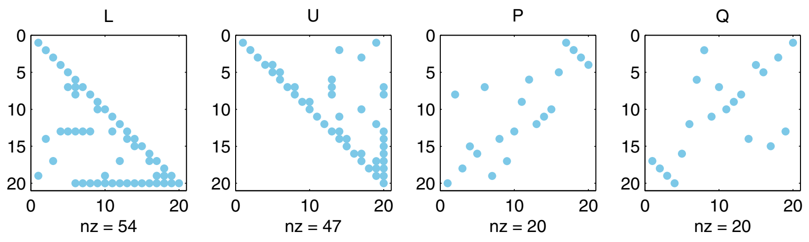


Figure 5.12. Matrices L, U, P and Q of the total pivoting factorization of the matrix A given in Figure 5.10, left

reordering the rows and columns of A, as we can grasp by inspecting the pattern of the permutation matrices P and Q.

In Figure 5.12 we report the matrices L, U, P, and Q obtained by the factorization of the matrix A of Figure 5.10. The fill-in generated by the total pivoting is much lighter than the one that would be produced if using partial pivoting by rows.

See Exercises 5.6-5.8.

5.5 How accurate is the solution of a linear system?

We have already noticed in Example 5.8 that, due to roundoff errors, the product LU does not reproduce A exactly. However, in general the pivoting allows keeping these errors under control and therefore achieving accurate solutions. (From the theoretical point of view, total pivoting is more stable than partial pivoting by rows, nevertheless experience shows that in general the latter provides accurate results by itself (see, e.g., [Hig02, Sez. 9.3]).)

Unfortunately, this is not always true, as the following example shows.

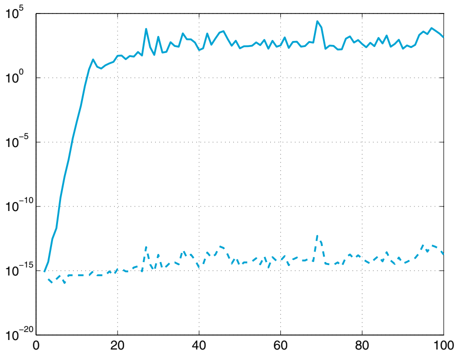


Figure 5.13. Behavior versus n of E_n (*solid line*) and of $\max_{i,j=1,\ldots,n}|r_{ij}|$ (*dashed line*) in logarithmic scale, for the Hilbert system of Example 5.9. The r_{ij} are the coefficients of the matrix R_n

Example 5.9 Consider the linear system $\mathrm{A}_n\mathbf{x}_n = \mathbf{b}_n$, where $\mathrm{A}_n \in \mathbb{R}^{n\times n}$ is the so-called *Hilbert matrix* whose elements are

$$a_{ij} = 1/(i+j-1), \qquad i,j = 1,\ldots,n,$$

while $\mathbf{b}_n$ is chosen in such a way that the exact solution is $\mathbf{x}_n = (1,1,\ldots,1)^T$. The matrix A_n is clearly symmetric and one can prove that it is also positive definite. For different values of n we use the MATLAB function `lu` to get the LU factorization of A_n with pivoting by row. Then we solve the associated linear systems (5.22) and denote by $\widehat{\mathbf{x}}_n$ the computed solution. In Figure 5.13 we report (in logarithmic scale) the relative errors

$$E_n = \|\mathbf{x}_n - \widehat{\mathbf{x}}_n\|/\|\mathbf{x}_n\|, \tag{5.26}$$

having denoted by $\|\cdot\|$ the Euclidean norm introduced in the Section 1.4.1. We have $E_n \geq 10$ if $n \geq 13$ (that is a relative error on the solution higher than 1000%!), whereas $\mathrm{R}_n = \mathrm{L}_n\mathrm{U}_n - \mathrm{P}_n\mathrm{A}_n$ is the null matrix (up to machine accuracy) for any given value of n. Similar results are obtained by using total pivoting. ■

On the ground of the previous remark, we could speculate by saying that, when a linear system $\mathrm{A}\mathbf{x} = \mathbf{b}$ is solved numerically, one is indeed looking for the *exact* solution $\widehat{\mathbf{x}}$ of a *perturbed* system

$$(\mathrm{A} + \delta\mathrm{A})\widehat{\mathbf{x}} = \mathbf{b} + \boldsymbol{\delta}\mathbf{b}, \tag{5.27}$$

where $\delta\mathrm{A}$ and $\boldsymbol{\delta}\mathbf{b}$ are respectively a matrix and a vector which depend on the specific numerical method which is being used. We start by considering the case where $\delta\mathrm{A} = 0$ and $\boldsymbol{\delta}\mathbf{b} \neq \mathbf{0}$ which is simpler than the most general case. Moreover, for simplicity we will also assume that $\mathrm{A} \in \mathbb{R}^{n\times n}$ is symmetric and positive definite.

By comparing (5.1) and (5.27) we find $\mathbf{x} - \widehat{\mathbf{x}} = -\mathrm{A}^{-1}\boldsymbol{\delta}\mathbf{b}$, and thus

$$\|\mathbf{x}-\widehat{\mathbf{x}}\| = \|\mathrm{A}^{-1}\boldsymbol{\delta}\mathbf{b}\|. \tag{5.28}$$

In order to find an upper bound for the right-hand side of (5.28), we proceed as follows. Since A is symmetric and positive definite, the set of its eigenvectors $\{\mathbf{v}_i\}_{i=1}^n$ provides an orthonormal basis of $\mathbb{R}^n$ (see [QSS07, Chapter 5]). This means that

$$\mathrm{A}\mathbf{v}_i = \lambda_i \mathbf{v}_i,\ i=1,\ldots,n, \qquad \mathbf{v}_i^T\mathbf{v}_j = \delta_{ij},\ i,j=1,\ldots,n,$$

where λ_i is the eigenvalue of A associated with $\mathbf{v}_i$ and δ_{ij} is the Kronecker symbol. Consequently, a generic vector $\mathbf{w}\in\mathbb{R}^n$ can be written as

$$\mathbf{w} = \sum_{i=1}^n w_i \mathbf{v}_i,$$

for a suitable (and unique) set of coefficients $w_i\in\mathbb{R}$. We have

$$\begin{aligned}\|\mathrm{A}\mathbf{w}\|^2 &= (\mathrm{A}\mathbf{w})^T(\mathrm{A}\mathbf{w})\\ &= [w_1(\mathrm{A}\mathbf{v}_1)^T + \ldots + w_n(\mathrm{A}\mathbf{v}_n)^T][w_1\mathrm{A}\mathbf{v}_1 + \ldots + w_n\mathrm{A}\mathbf{v}_n]\\ &= (\lambda_1 w_1\mathbf{v}_1^T + \ldots + \lambda_n w_n \mathbf{v}_n^T)(\lambda_1 w_1\mathbf{v}_1 + \ldots + \lambda_n w_n\mathbf{v}_n)\\ &= \sum_{i=1}^n \lambda_i^2 w_i^2.\end{aligned}$$

Denote by λ_{max} the largest eigenvalue of A. Since $\|\mathbf{w}\|^2 = \sum_{i=1}^n w_i^2$, we conclude that

$$\|\mathrm{A}\mathbf{w}\| \le \lambda_{max}\|\mathbf{w}\| \quad \forall\mathbf{w}\in\mathbb{R}^n. \tag{5.29}$$

In a similar manner, we obtain

$$\|\mathrm{A}^{-1}\mathbf{w}\| \le \frac{1}{\lambda_{min}}\|\mathbf{w}\|,$$

upon recalling that the eigenvalues of A^{-1} are the reciprocals of those of A. This inequality enables us to draw from (5.28) that

$$\frac{\|\mathbf{x}-\widehat{\mathbf{x}}\|}{\|\mathbf{x}\|} \le \frac{1}{\lambda_{min}}\frac{\|\boldsymbol{\delta}\mathbf{b}\|}{\|\mathbf{x}\|}. \tag{5.30}$$

Using (5.29) once more and recalling that $\mathrm{A}\mathbf{x}=\mathbf{b}$, we finally obtain

$$\boxed{\frac{\|\mathbf{x}-\widehat{\mathbf{x}}\|}{\|\mathbf{x}\|} \le \frac{\lambda_{max}}{\lambda_{min}}\frac{\|\boldsymbol{\delta}\mathbf{b}\|}{\|\mathbf{b}\|}} \tag{5.31}$$

We can conclude that the relative error in the solution depends on the relative error in the data through the following constant (≥ 1)

$$\boxed{K(\mathrm{A}) = \frac{\lambda_{max}}{\lambda_{min}}} \tag{5.32}$$

which is called *spectral condition number of the matrix* A. $K(\mathrm{A})$ can be computed in MATLAB using the command `cond`.

`cond`

Remark 5.4 The MATLAB command `cond(A)` allows the computation of the condition number of any type of matrix `A`, even those which are not symmetric and positive definite. It is worth mentioning that there exist various definitions of condition number of a matrix. For a generic matrix A, the command `cond(A)` computes the value $K_2(\mathrm{A}) = \|\mathrm{A}\|_2 \cdot \|\mathrm{A}^{-1}\|_2$, where we define $\|\mathrm{A}\|_2 = \sqrt{\lambda_{max}(\mathrm{A}^T\mathrm{A})}$. We note that if A is not symmetric and positive definite, $K_2(\mathrm{A})$ can be very far from the spectral condition number $K(\mathrm{A})$. For a sparse matrix A, the command `condest(A)` computes an approximation (at low computational cost) of the condition number $K_1(\mathrm{A}) = \|\mathrm{A}\|_1 \cdot \|\mathrm{A}^{-1}\|_1$, being $\|\mathrm{A}\|_1 = \max_j \sum_{i=1}^{n} |a_{ij}|$ the so-called *1-norm* of A. Other definitions for the condition number are available for nonsymmetric matrices, see [QSS07, Chapter 3]. ■

`condest`

A more involved proof would lead to the following more general result in the case where A is symmetric and positive definite and $\delta\mathrm{A}$ is an arbitrary symmetric and positive definite matrix, "small enough" to satisfy $\lambda_{max}(\delta\mathrm{A}) < \lambda_{min}(\mathrm{A})$:

$$\boxed{\frac{\|\mathbf{x}-\widehat{\mathbf{x}}\|}{\|\mathbf{x}\|} \le \frac{K(\mathrm{A})}{1-\lambda_{max}(\delta\mathrm{A})/\lambda_{min}(\mathrm{A})}\left(\frac{\lambda_{max}(\delta\mathrm{A})}{\lambda_{max}(\mathrm{A})} + \frac{\|\boldsymbol{\delta}\mathbf{b}\|}{\|\mathbf{b}\|}\right)} \tag{5.33}$$

Finally, if A and $\delta\mathrm{A}$ are not symmetric positive definite matrices, and $\delta\mathrm{A}$ is such that $\|\delta\mathrm{A}\|_2\,\|\mathrm{A}^{-1}\|_2 < 1$, the following estimate holds:

$$\boxed{\frac{\|\mathbf{x}-\widehat{\mathbf{x}}\|}{\|\mathbf{x}\|} \le \frac{K_2(\mathrm{A})}{1-K_2(\mathrm{A})\|\delta\mathrm{A}\|_2/\|\mathrm{A}\|_2}\left(\frac{\|\delta\mathrm{A}\|_2}{\|\mathrm{A}\|_2} + \frac{\|\boldsymbol{\delta}\mathbf{b}\|}{\|\mathbf{b}\|}\right)} \tag{5.34}$$

If $K(\mathrm{A})$ is "small", that is of the order of unity, A is said to be *well conditioned*. In that case, small errors in the data will lead to errors of the same order of magnitude in the solution. This would not occur in the case of *ill conditioned* matrices.

Example 5.10 For the Hilbert matrix introduced in Example 5.9, $K(\mathrm{A}_n)$ is a rapidly increasing function of n. One has $K(\mathrm{A}_4) > 15000$, while if $n > 13$ the condition number is so high that MATLAB warns that the matrix is "close to singular". Actually, $K(\mathrm{A}_n)$ grows at an exponential rate, $K(\mathrm{A}_n) \simeq e^{3.5n}$ (see, [Hig02]). This provides an indirect explanation of the bad results obtained in Example 5.9. ■

Inequality (5.31) can be reformulated by the help of the *residual* $\mathbf{r}$

$$\mathbf{r} = \mathbf{b} - \mathrm{A}\widehat{\mathbf{x}}. \tag{5.35}$$

Should $\widehat{\mathbf{x}}$ be the exact solution, the residual would be the null vector. Thus, in general, $\mathbf{r}$ can be regarded as an *estimator* of the error $\mathbf{x} - \widehat{\mathbf{x}}$. The extent to which the residual is a good error estimator depends on the size of the condition number of A. Indeed, observing that $\boldsymbol{\delta}\mathbf{b} = \mathrm{A}(\widehat{\mathbf{x}} - \mathbf{x}) = \mathrm{A}\widehat{\mathbf{x}} - \mathbf{b} = -\mathbf{r}$, we deduce from (5.31) that

$$\boxed{\frac{\|\mathbf{x} - \widehat{\mathbf{x}}\|}{\|\mathbf{x}\|} \leq K(\mathrm{A})\frac{\|\mathbf{r}\|}{\|\mathbf{b}\|}} \tag{5.36}$$

Thus if $K(\mathrm{A})$ is "small", we can be sure that the error is small provided that the residual is small, whereas this might not be true when $K(\mathrm{A})$ is "large".

Example 5.11 The residuals associated with the computed solution of the linear systems of Example 5.9 are very small (their norms vary between 10^{-16} and 10^{-11}); however the computed solutions differ remarkably from the exact solution. ■

See Exercises 5.9-5.10.

5.6 How to solve a tridiagonal system

In many applications (see for instance Chapter 9), we have to solve a system whose matrix has the form

$$\mathrm{A} = \begin{bmatrix} a_1 & c_1 & & 0 \\ e_2 & a_2 & \ddots & \\ & \ddots & & c_{n-1} \\ 0 & & e_n & a_n \end{bmatrix}.$$

This matrix is called *tridiagonal* since the only elements that can be non-null belong to the main diagonal and to the first super and sub diagonals.

If the LU factorization of A exists, the factors L and U must be *bidiagonals* (lower and upper, respectively), more precisely:

$$\mathrm{L} = \begin{bmatrix} 1 & & 0 \\ \beta_2 & 1 & \\ & \ddots & \ddots & \\ 0 & & \beta_n & 1 \end{bmatrix}, \quad \mathrm{U} = \begin{bmatrix} \alpha_1 & c_1 & & 0 \\ & \alpha_2 & \ddots & \\ & & \ddots & c_{n-1} \\ 0 & & & \alpha_n \end{bmatrix}.$$

The unknown coefficients α_i and β_i can be determined by requiring that the equality LU = A holds. This yields the following recursive relations for the computation of the L and U factors:

$$\alpha_1 = a_1, \quad \beta_i = \frac{e_i}{\alpha_{i-1}}, \quad \alpha_i = a_i - \beta_i c_{i-1}, \quad i = 2, \ldots, n. \tag{5.37}$$

Using (5.37), we can easily solve the two bidiagonal systems $\mathrm{L}\mathbf{y} = \mathbf{b}$ and $\mathrm{U}\mathbf{x} = \mathbf{y}$, to obtain the following formulae:

$$(\mathrm{L}\mathbf{y} = \mathbf{b}) \quad y_1 = b_1, \quad y_i = b_i - \beta_i y_{i-1}, \quad i = 2, \ldots, n, \tag{5.38}$$

$$(\mathrm{U}\mathbf{x} = \mathbf{y}) \quad x_n = \frac{y_n}{\alpha_n}, \;\; x_i = (y_i - c_i x_{i+1})\,/\alpha_i, \;\; i = n-1, \ldots, 1. \tag{5.39}$$

This is known as the *Thomas algorithm* and allows the solution of the original system with a computational cost of the order of n operations.

The MATLAB command `spdiags` allows the construction of a tridiagonal matrix by storing only the non-null diagonals. For instance, the commands

```
b=ones(10,1);
a=2*b;
c=3*b;
T=spdiags([b a c],-1:1,10,10);
```

compute the tridiagonal matrix $\mathrm{T} \in \mathbb{R}^{10\times 10}$ with elements equal to 2 on the main diagonal, 1 on the first subdiagonal and 3 on the first superdiagonal.

Note that T is stored in a *sparse mode*, according to which the only elements stored are those different than 0. When A is a tridiagonal matrix generated in sparse mode, the Thomas algorithm is the solution algorithm selected by the MATLAB command \. (See also Section 5.8 for a more general discussion on the MATLAB command \.)

5.7 Overdetermined systems

A linear system $\mathrm{A}\mathbf{x}$=$\mathbf{b}$ with A$\in \mathbb{R}^{m\times n}$ is called *overdetermined* if $m > n$, *underdetermined* if $m < n$.

An overdetermined system generally has no solution unless the right hand side vector $\mathbf{b}$ is an element of range(A), where

$$\mathrm{range(A)} = \{\mathbf{z} \in \mathbb{R}^m : \; \mathbf{z} = \mathrm{A}\mathbf{y} \text{ for } \mathbf{y} \in \mathbb{R}^n\}. \tag{5.40}$$

In general, for an arbitrary $\mathbf{b}$ we can search a vector $\mathbf{x}^* \in \mathbb{R}^n$ that minimizes the Euclidean norm of the residual, that is,

$$\Phi(\mathbf{x}^*) = \|\mathrm{A}\mathbf{x}^* - \mathbf{b}\|_2^2 \leq \|\mathrm{A}\mathbf{y} - \mathbf{b}\|_2^2 = \Phi(\mathbf{y}) \qquad \forall \mathbf{y} \in \mathbb{R}^n. \tag{5.41}$$

When it does exist, the vector $\mathbf{x}^*$ is called *least-squares solution* of the overdetermined system $\mathrm{A}\mathbf{x}=\mathbf{b}$.

Similarly to what was done in Section 3.6, the solution of (5.41) can be found by imposing the condition that the gradient of the function Φ must be equal to zero at $\mathbf{x}^*$. With similar calculations we find that $\mathbf{x}^*$ is in fact the solution of the square $n \times n$ linear system

$$\boxed{\mathrm{A}^T\mathrm{A}\mathbf{x}^* = \mathrm{A}^T\mathbf{b}} \tag{5.42}$$

which is called the system of *normal equations*. The system (5.42) is nonsingular if A has *full rank* (that is rank(A) = min(m,n), where the *rank* of A, rank(A), is the maximum order of the nonvanishing determinants extracted from A). In such a case $\mathrm{B} = \mathrm{A}^T\mathrm{A}$ is a symmetric and positive definite matrix, then the least-squares solution exists and is unique.

To compute it one could use the Cholesky factorization (5.17) applied to the matrix B. However, due to roundoff errors, the computation of $\mathrm{A}^T\mathrm{A}$ may be affected by a loss of significant digits, with a consequent loss of the positive definiteness of the matrix itself. Instead, it is more convenient to use either the so-called QR factorization of A, or the Singular Value Decomposition (SVD) of A.

Let us start from the former. Any full rank matrix $\mathrm{A} \in \mathbb{R}^{m\times n}$, with $m \geq n$, admits a unique *QR factorization*

$$\boxed{\mathrm{A} = \mathrm{QR}} \tag{5.43}$$

$\mathrm{Q} \in \mathbb{R}^{m\times m}$ is an orthogonal matrix (i.e. $\mathrm{Q}^T\mathrm{Q} = \mathrm{I}$), while $\mathrm{R} \in \mathbb{R}^{m\times n}$ is a rectangular matrix whose entries below the main diagonal are equal to zero, whereas all its diagonal entries are non-null. See Figure 5.14.

It is possible to prove that $\mathrm{A} = \widetilde{\mathrm{Q}}\widetilde{\mathrm{R}}$, where $\widetilde{\mathrm{Q}} = \mathrm{Q}(1:m, 1:n)$ and $\widetilde{\mathrm{R}} = \mathrm{R}(1:n, 1:n)$ are the submatrices indicated in Figure 5.14. $\widetilde{\mathrm{Q}}$ has orthonormal column vectors, while $\widetilde{\mathrm{R}}$ is an upper triangular matrix, which in fact coincides with the triangular factor R of Cholesky factorization of the matrix $\mathrm{A}^T\mathrm{A}$. Since $\widetilde{\mathrm{R}}$ is non-singular, the unique solution of (5.43) reads

$$\mathbf{x}^* = \tilde{\mathrm{R}}^{-1}\tilde{\mathrm{Q}}^T\mathbf{b}. \tag{5.44}$$

Now let us turn to the singular value decomposition of a matrix: for any given rectangular matrix $\mathrm{A} \in \mathbb{C}^{m\times n}$, there exist two unitary matrices $\mathrm{U} \in \mathbb{C}^{m\times m}$ and $\mathrm{V} \in \mathbb{C}^{n\times n}$, such that

$$\mathrm{U}^H\mathrm{A}\mathrm{V} = \Sigma = \mathrm{diag}(\sigma_1, \ldots, \sigma_p) \in \mathbb{R}^{m\times n} \tag{5.45}$$

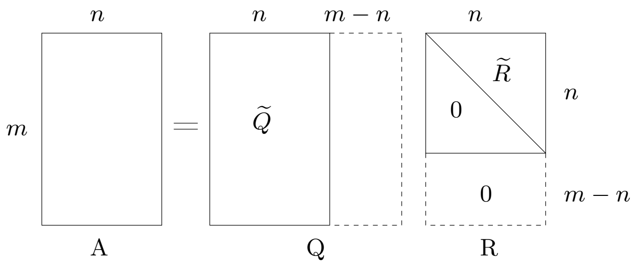


Figure 5.14. The QR factorization

where $p = \min(m,n)$ and $\sigma_1 \geq \ldots \geq \sigma_p \geq 0$. A matrix U is said *unitary* if $\mathrm{U}^H\mathrm{U} = \mathrm{U}\mathrm{U}^H = \mathrm{I}$. Formula (5.45) is named *singular value decomposition* (SVD in short) of A and the entries σ_i of Σ are named *singular values* of A. It holds that $\sigma_i = \sqrt{\lambda_i(\mathrm{A}^H\mathrm{A})}$, while $\lambda_i(\mathrm{A}^H\mathrm{A})$ are the real positive eigenvalues of the matrix $\mathrm{A}^H\mathrm{A}$.

If A is a real matrix, then also U and V are real matrices. Moreover, U and V are *orthogonal* matrices and U^H coincides with U^T.

Let us now compute the singular value decomposition (5.45) of the matrix A in (5.42). Since U is orthogonal, $\mathrm{A}^T\mathrm{A} = \mathrm{V}\Sigma^T\Sigma V^T$, hence the system of normal equations (5.42) is equivalent to the system

$$\mathrm{V}\Sigma^T\Sigma\mathrm{V}^T\mathbf{x}^* = \mathrm{V}\Sigma^T\mathrm{U}^T\mathbf{b}. \tag{5.46}$$

We note that also V is orthogonal and that $\Sigma^T\Sigma$ is a square non-singular matrix whose diagonal entries are the square of the singular values of A. Therefore, by a left multiplication of equation (5.46) by $\mathrm{V}(\Sigma^T\Sigma)^{-1}V^T$ we have

$$\mathbf{x}^* = \mathrm{V}\Sigma^\dagger U^T\mathbf{b} = \mathrm{A}^\dagger\mathbf{b}, \tag{5.47}$$

where $\Sigma^\dagger = (\Sigma^T\Sigma)^{-1}\Sigma^T = \mathrm{diag}(1/\sigma_1,\ldots,1/\sigma_n,0,\ldots,0)$ and $A^\dagger = (\mathrm{A}^T\mathrm{A})^{-1}\mathrm{A}^T = \mathrm{V}\Sigma^\dagger U^T$. The latter matrix is called *pseudoinverse* of A.

We deduce from formula (5.47) that after computing the singular values of A and the matrices U and V, by a small additional effort we can find the solution of the normal equations (5.42).

Two functions are available in MATLAB for the computation of the SVD of a given matrix: `svd` and `svds`. The former computes all singular values of A, the latter only the largest `k` singular values, where `k` is a parameter given in input (the default value is `k=6`). We refer to [ABB+99] for an exhaustive description of algorithms used in MATLAB. `svd` `svds`

Example 5.12 Consider an alternative approach to the problem of finding the regression line $\epsilon(\sigma) = a_1\sigma + a_0$ (see Section 3.6) of the data of Problem 3.3. Using the data of Table 3.2 and imposing the interpolating conditions we obtain the overdetermined system $\mathtt{A}\mathbf{a} = \mathtt{b}$, where $\mathbf{a} = (a_1, a_0)^T$ and

$$\mathtt{A} = \begin{bmatrix} 0 & 1 \\ 0.06 & 1 \\ 0.14 & 1 \\ 0.25 & 1 \\ 0.31 & 1 \\ 0.47 & 1 \\ 0.60 & 1 \\ 0.70 & 1 \end{bmatrix}, \quad \mathtt{b} = \begin{bmatrix} 0 \\ 0.08 \\ 0.14 \\ 0.20 \\ 0.23 \\ 0.25 \\ 0.28 \\ 0.29 \end{bmatrix}.$$

In order to compute its least-squares solution we use the following instructions

```
[Q,R]=qr(A);
Qt=Q(:,1:2); Rt=R(1:2,:);
xstar = Rt \ (Qt'*b)

xstar =
    0.3741
    0.0654
```

These are precisely the same coefficients for the regression line computed in the Example 3.13. Notice that this procedure is directly implemented in the command \: in fact, the instruction `xstar = A\b` produces the same `xstar` vector, computed by the formulae (5.43) and (5.44). ■

5.8 What is hidden behind the MATLAB command \

It is useful to know that the specific algorithm used by MATLAB when the \ command is invoked depends upon the structure of the matrix A. To determine the structure of A and select the appropriate algorithm, MATLAB follows this precedence:

1. if A is sparse and banded, then banded solvers are used (like the Thomas algorithm of Section 5.6);
2. if A is an upper or lower triangular matrix (or else a permutation of a triangular matrix), then the system is solved by a backward substitution algorithm for upper triangular matrices, or by a forward substitution algorithm for lower triangular matrices. The check for triangularity is done for full matrices by testing for zero elements and for sparse matrices by accessing the sparse data structure;
3. if A is symmetric and has real positive diagonal elements (which does not imply that A is positive definite), then a Cholesky factorization is attempted (`chol`). If A is sparse, a preordering algorithm is applied first;
4. if none of previous criteria are fulfilled, then a general triangular factorization is computed by Gaussian elimination with partial pivoting (`lu`);
5. if A is sparse, then the UMFPACK library (which is part of the Suitesparse suite, see e.g. `http://www.cise.ufl.edu/research/sparse/SuiteSparse/`) is used to compute the solution of the system;

6. if A is not square, proper methods based on the QR factorization for undetermined systems are used (for the overdetermined case, see Section 5.7).

The command \ is also available in Octave. On a system with dense matrix, Octave operates as follows:

1. if the matrix is upper (resp., lower) triangular, Octave call backward (resp., forward) substitutions of LAPACK (a widely used library of linear algebra routines [ABB+99]);
2. if the matrix is symmetric and has real positive diagonal entries, Octave attempts a Cholesky factorization by LAPACK;
3. if either the Cholesky factorization fails or the matrix is not symmetric with positive diagonal entries, the system is solved by Gaussian elimination with pivoting by rows by LAPACK;
4. if the matrix is not square, or any of the previous solvers flags a singular or near singular matrix, Octave looks for a solution in the least-squares sense.

For a linear system with sparse matrix, like MATLAB Octave relies on UMFPACK and other packages from the Suitesparse collection, in particular:

1. for a square, banded matrix with "small enough" band density continue to a), else goto 2;
   a) if the matrix is tridiagonal and the right-hand side is not sparse continue, else goto b);
      i. if the matrix is symmetric with positive diagonal entries, Octave attempts a Cholesky factorization;
      ii. if the above failed or the matrix is not symmetric with positive diagonal entries, then it uses Gaussian elimination with pivoting;
   b) if the matrix is symmetric with positive diagonal entries, Octave attempts a Cholesky factorization;
   c) if the above failed or the matrix is not symmetric with positive diagonal entries, Octave uses Gaussian elimination with pivoting;
2. if the matrix is upper (with column permutations) or lower (with row permutations) triangular, perform a sparse forward or backward substitution;
3. if the matrix is square, symmetric with positive diagonal entries, Octave attempts sparse Cholesky factorization;
4. if the sparse Cholesky factorization failed or the matrix is not symmetric with positive diagonal entries, Octave factorizes the matrix using the UMFPACK library;
5. if the matrix is not square, or any of the previous solvers flags a singular or near singular matrix, Octave provides a solution in the least-squares sense.

Let us summarize

1. The LU factorization of A$\in\mathbb{R}^{n\times n}$ consists in computing a lower triangular matrix L and an upper triangular matrix U such that A = LU;
2. the LU factorization, provided it exists, is not unique. However, it can be determined unequivocally by providing an additional condition such as, e.g., setting the diagonal elements of L equal to 1. This is called LU factorization;
3. the LU factorization exists and is unique if and only if the principal submatrices of A of order 1 to $n-1$ are nonsingular (otherwise at least one pivot element is null);
4. if a null pivot element is generated, a new pivot element can be obtained by exchanging in a suitable manner two rows (or columns) of our system. This is the pivoting strategy;
5. the computation of the LU factorization requires about $2n^3/3$ operations, and only an order of n operations in the case of tridiagonal systems;
6. for symmetric and positive definite matrices we can use the Cholesky factorization A = R^{T}R, where R is an upper triangular matrix, and the computational cost is of the order of $n^3/3$ operations;
7. the sensitivity of the result to perturbation of data depends on the condition number of the system matrix; more precisely, the accuracy of the computed solution can be low for ill conditioned matrices;
8. the solution of an overdetermined linear system can be intended in the least-squares sense and can be computed using either QR factorization or singular value decomposition (SVD).

5.9 Iterative methods

Let us consider the linear system (5.1) with A$\in\mathbb{R}^{n\times n}$ and $\mathbf{b}\in\mathbb{R}^n$. An iterative method for the solution of (5.1) consists in setting up a sequence of vectors $\{\mathbf{x}^{(k)}, k\geq 0\}$ of $\mathbb{R}^n$ that *converges* to the exact solution $\mathbf{x}$, that is

$$\lim_{k\to\infty}\mathbf{x}^{(k)}=\mathbf{x}, \tag{5.48}$$

for any given initial vector $\mathbf{x}^{(0)}\in\mathbb{R}^n$. A possible strategy able to realize this process can be based on the following recursive definition

$$\mathbf{x}^{(k+1)}=\mathrm{B}\mathbf{x}^{(k)}+\mathbf{g},\qquad k\geq 0, \tag{5.49}$$

where B is a suitable matrix (depending on A) and $\mathbf{g}$ is a suitable vector (depending on A and $\mathbf{b}$), which must satisfy the consistency relation

$$\mathbf{x} = \mathrm{B}\mathbf{x} + \mathbf{g}. \tag{5.50}$$

Since $\mathbf{x} = \mathrm{A}^{-1}\mathbf{b}$ this yields $\mathbf{g} = (\mathrm{I} - \mathrm{B})\mathrm{A}^{-1}\mathbf{b}$.

Let $\mathbf{e}^{(k)} = \mathbf{x} - \mathbf{x}^{(k)}$ define the error at step k. By subtracting (5.49) from (5.50), we obtain

$$\mathbf{e}^{(k+1)} = \mathrm{B}\mathbf{e}^{(k)}.$$

For this reason B is called the *iteration matrix* associated with (5.49). If B is symmetric and positive definite, by (5.29) we have

$$\|\mathbf{e}^{(k+1)}\| = \|\mathrm{B}\mathbf{e}^{(k)}\| \leq \rho(\mathrm{B})\|\mathbf{e}^{(k)}\|, \qquad k \geq 0.$$

We have denoted by $\rho(\mathrm{B})$ the *spectral radius* of B, that is, the maximum modulus of eigenvalues of B. If B is a symmetric positive definite matrix, then $\rho(\mathrm{B})$ coincides with the largest eigenvalue of B. By iterating the same inequality backward, we obtain

$$\|\mathbf{e}^{(k)}\| \leq [\rho(\mathrm{B})]^k \|\mathbf{e}^{(0)}\|, \quad k \geq 0. \tag{5.51}$$

Thus $\mathbf{e}^{(k)} \to \mathbf{0}$ as $k \to \infty$ for every possible $\mathbf{e}^{(0)}$ (and henceforth $\mathbf{x}^{(0)}$) provided that $\rho(\mathrm{B}) < 1$. Therefore, the method converges. Actually, this property is also necessary for convergence.

Should, by any chance, an approximate value of $\rho(\mathrm{B})$ be available, (5.51) would allow us to deduce the minimum number of iterations k_{min} that are needed to damp the initial error by a factor ε. Indeed, k_{min} would be the lowest positive integer for which $[\rho(\mathrm{B})]^{k_{min}} \leq \varepsilon$.

In conclusion, for a generic matrix the following result holds:

Proposition 5.2 *For an iterative method of the form* (5.49) *whose iteration matrix satisfies* (5.50)*, convergence for any* $\mathbf{x}^{(0)}$ *holds iff* $\rho(\mathrm{B}) < 1$*. Moreover, the smaller* $\rho(\mathrm{B})$*, the fewer the number of iterations necessary to reduce the initial error by a given factor.*

5.9.1 How to construct an iterative method

A general technique to devise an iterative method is based on a *splitting* of the matrix A, $\mathrm{A} = \mathrm{P} - (\mathrm{P} - \mathrm{A})$, being P a suitable nonsingular matrix (called the *preconditioner* of A). Then

$$\mathrm{P}\mathbf{x} = (\mathrm{P} - \mathrm{A})\mathbf{x} + \mathbf{b},$$

has the form (5.50) provided that we set $\mathrm{B} = \mathrm{P}^{-1}(\mathrm{P} - \mathrm{A}) = \mathrm{I} - \mathrm{P}^{-1}\mathrm{A}$ and $\mathbf{g} = \mathrm{P}^{-1}\mathbf{b}$. Correspondingly, we can define the following iterative method

$$\mathrm{P}(\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}) = \mathbf{r}^{(k)}, \qquad k \geq 0,$$

where

$$\boxed{\mathbf{r}^{(k)} = \mathbf{b} - \mathrm{A}\mathbf{x}^{(k)}} \tag{5.52}$$

denotes the residual vector at iteration k. A generalization of this iterative method is the following

$$\boxed{\mathrm{P}(\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}) = \alpha_k \mathbf{r}^{(k)}, \qquad k \geq 0} \tag{5.53}$$

where $\alpha_k \neq 0$ is a parameter that may change at every iteration k and which, a priori, will be useful to improve the convergence properties of the sequence $\{\mathbf{x}^{(k)}\}$.

The method (5.53) requires to find at each step the so-called *preconditioned residual* $\mathbf{z}^{(k)}$ which is the solution of the linear system

$$\mathrm{P}\mathbf{z}^{(k)} = \mathbf{r}^{(k)}, \tag{5.54}$$

then the new iterate is defined by $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{z}^{(k)}$. For that reason the matrix P ought to be chosen in such a way that the computational cost for the solution of (5.54) be quite low (e.g., every P either diagonal or triangular or tridiagonal will serve the purpose). Let us now consider some special instance of iterative methods which take the form (5.53).

The Jacobi method

If the diagonal entries of A are nonzero, we can set $\mathrm{P} = \mathrm{D} = \mathrm{diag}(a_{11}, a_{22}, \ldots, a_{nn})$, that is D is the diagonal matrix containing the diagonal entries of A. The Jacobi method corresponds to this choice with the assumption $\alpha_k = 1$ for all k. Then from (5.53) we obtain

$$\mathrm{D}\mathbf{x}^{(k+1)} = \mathbf{b} - (\mathrm{A} - \mathrm{D})\mathbf{x}^{(k)}, \qquad k \geq 0,$$

or, componentwise,

$$\boxed{x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1, j\neq i}^{n} a_{ij} x_j^{(k)} \right), \ i = 1, \ldots, n} \tag{5.55}$$

where $k \geq 0$ and $\mathbf{x}^{(0)} = (x_1^{(0)}, x_2^{(0)}, \ldots, x_n^{(0)})^T$ is the initial vector.

The iteration matrix is therefore

$$\mathrm{B} = \mathrm{D}^{-1}(\mathrm{D} - \mathrm{A}) = \begin{bmatrix} 0 & -a_{12}/a_{11} & \ldots & -a_{1n}/a_{11} \\ -a_{21}/a_{22} & 0 & & -a_{2n}/a_{22} \\ \vdots & & \ddots & \vdots \\ -a_{n1}/a_{nn} & -a_{n2}/a_{nn} & \ldots & 0 \end{bmatrix}. \tag{5.56}$$

The following result allows the verification of Proposition 5.2 without explicitly computing $\rho(\mathrm{B})$:

Proposition 5.3 *If the matrix* A$\in \mathbb{R}^{n\times n}$ *of system (5.1) is strictly diagonally dominant by row, then the Jacobi method converges.*

As a matter of fact, we can verify that $\rho(\mathrm{B}) < 1$, where B is given in (5.56), that is, all eigenvalues of B are in modulus less than 1. To start with, we note that the diagonal elements of A are non-null owing to the strict diagonal dominance (see Section 6.4). Let λ be a generic eigenvalue of B and $\mathbf{x}$ an associated eigenvector. Then

$$\sum_{j=1}^{n} b_{ij}x_j = \lambda x_i,\ i=1,\ldots,n.$$

Assume for simplicity that $\max_{k=1,\ldots,n}|x_k| = 1$ (this is not restrictive since an eigenvector is defined up to a multiplicative constant) and let x_i be the component whose modulus is equal to 1. Then

$$|\lambda| = \left|\sum_{j=1}^{n} b_{ij}x_j\right| = \left|\sum_{j=1, j\neq i}^{n} b_{ij}x_j\right| \leq \sum_{j=1, j\neq i}^{n} \left|\frac{a_{ij}}{a_{ii}}\right|,$$

having noticed that B has only null diagonal elements. Therefore $|\lambda| < 1$ thanks to the assumption made on A.

The Jacobi method is implemented in the Program 5.2 setting in the input parameter `P='J'`. Other input parameters are: the system matrix `A`, the right hand side `b`, the initial vector `x0`, the maximum number of iterations allowed, `nmax` and a given tolerance `tol` for stopping test. The iterative procedure is terminated as soon as the ratio between the Euclidean norm of the current residual and that of the initial residual is less than or equal to `tol` (for a justification of this stopping criterion, see Section 5.12).

Program 5.2. itermeth: general iterative method

```
function [x,iter]= itermeth(A,b,x0,nmax,tol,P)
%ITERMETH  General iterative method
% X = ITERMETH(A,B,X0,NMAX,TOL,P) attempts to solve the
% system of linear equations A*X=B for X. The N-by-N
% coefficient matrix A must be non-singular and the
% right hand side column vector B must have length
% N. If P='J' the Jacobi method is used, if P='G' the
% Gauss-Seidel method is selected. Otherwise, P is a
% N-by-N matrix that plays the role of a preconditioner
% for the gradient method, which is a dynamic
% Richardson method. Iterations
% stop when the ratio between the norm of the kth
% residual and the norm of the initial residual is less
% than TOL, then ITER is the number of performed
% iterations. NMAX specifies the maximum
% number of iterations. If P is not defined, the
% unpreconditioned gradient method is performed.
[n,n]=size(A);
if nargin == 6
 if ischar(P)==1
  if P=='J'
   L=diag(diag(A)); U=eye(n); beta=1; alpha=1;
  elseif P == 'G'
   L=tril(A); U=eye(n); beta=1; alpha=1;
  end
 else
     [L,U]=lu(P); beta = 0;
 end
else
  L = eye(n); U = L; beta = 0;
end
iter=0; x=x0; r=b-A*x0;  r0=norm(r);  err=r0;
while err > tol & iter < nmax
  z = L\r;  z = U\z; iter = iter + 1;
  if beta == 0
     alpha = z'*r/(z'*A*z);
  end
  x = x + alpha*z;
  r = b - A * x;
  err = norm (r) / r0;
end
```

The Gauss-Seidel method

When applying the Jacobi method, each component $x_i^{(k+1)}$ of the new vector $\mathbf{x}^{(k+1)}$ is computed independently of the others. This may suggest that a faster convergence could be (hopefully) achieved if the new components already available $x_j^{(k+1)}$, $j = 1,\dots,i-1$, together with the old ones $x_j^{(k)}$, $j \geq i$, are used for the calculation of $x_i^{(k+1)}$. This would lead to modifying (5.55) as follows: for $k \geq 0$ (still assuming that $a_{ii} \neq 0$ for $i = 1,\dots,n$)

$$\boxed{x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}\right), i = 1,..,n} \quad (5.57)$$

The updating of the components is made in *sequential* mode, whereas in the original Jacobi method it is made *simultaneously* (or in parallel). The new method, which is called the *Gauss-Seidel method*, corresponds to the choice $\mathrm{P} = \mathrm{D} - \mathrm{E}$ and $\alpha_k = 1$, $k \geq 0$, in (5.53), where E is a lower triangular matrix whose non null entries are $e_{ij} = -a_{ij}$, $i = 2,\dots,n$, $j = 1,\dots,i-1$. The corresponding iteration matrix is then

$$\mathrm{B} = (\mathrm{D} - \mathrm{E})^{-1}(\mathrm{D} - \mathrm{E} - \mathrm{A}).$$

A possible generalization is the so-called in which $P = \frac{1}{\omega}D - E$, where $\omega \neq 0$ is the relaxation parameter, and $\alpha_k = 1$, $k \geq 0$ (see Exercise 5.13).

Also for the Gauss-Seidel method there exist special matrices A whose associated iteration matrices satisfy the assumptions of Proposition 5.2 (those guaranteeing convergence). Among them let us mention:

1. matrices which are strictly diagonally dominant by row;
2. matrices which are real symmetric and positive definite.

The Gauss-Seidel method is implemented in Program 5.2 setting the input parameter `P` equal to `'G'`.

There are no general results stating that the Gauss-Seidel method always converges faster than Jacobi's. However, in some special instances this is the case, as stated by the following proposition (see, e.g. [Saa03, Thm. 4.7]):

Proposition 5.4 *Let* $A \in \mathbb{R}^{n\times n}$ *be a tridiagonal nonsingular matrix whose diagonal elements are all non-null. Then the Jacobi method and the Gauss-Seidel method are either both divergent or both convergent. In the latter case, the Gauss-Seidel method is faster than Jacobi's; more precisely the spectral radius of its iteration matrix is equal to the square of that of Jacobi.*

Example 5.13 Let us consider a linear system $A\mathbf{x} = \mathbf{b}$, where $\mathbf{b}$ is chosen in such a way that the solution is the unit vector $(1, 1, \ldots, 1)^T$ and A is the 10×10 tridiagonal matrix whose diagonal entries are all equal to 3, the entries of the first lower diagonal are equal to -2 and those of the upper diagonal are all equal to -1. Both Jacobi and Gauss-Seidel methods converge since the spectral radii of their iteration matrices are strictly less than 1. More precisely, by starting from a null initial vector and setting `tol` $=10^{-12}$, the Jacobi method converges in 277 iterations while only 143 iterations are requested from Gauss-Seidel's. To get this result we have used the following instructions:

```
n=10;
A=3*eye(n)-2*diag(ones(n-1,1),1)-diag(ones(n-1,1),-1);
b=A*ones(n,1);
x0=zeros(n,1);
[x,iterJ]=itermeth(A,b,x0,400,1.e-12,'J'); iterJ
[x,iterG]=itermeth(A,b,x0,400,1.e-12,'G'); iterG

iterJ =
   277
iterG =
   143
```

■

See Exercises 5.11-5.14.

5.10 Richardson and gradient methods

Let us now reconsider a method that can be set in the general form (5.53). We call *stationary* the case when $\alpha_k = \alpha$ (a given constant) for any $k \geq 0$, *dynamic* the case in which α_k may change along the iterations. In this framework the nonsingular matrix P is still called a *preconditioner* of A.

The crucial issue is the way the parameters are chosen. In this respect, the following results hold (see, e.g., [QV94, Chapter 2], [Axe94]).

Proposition 5.5 *Let $A \in \mathbb{R}^{n\times n}$. For any non-singular matrix $\mathrm{P} \in \mathbb{R}^{n\times n}$ the stationary Richardson method converges iff*

$$|\lambda_i|^2 < \frac{2}{\alpha}\mathrm{Re}\lambda_i \qquad \forall i = 1, \ldots, n,$$

where λ_i are the eigenvalues of $\mathrm{P}^{-1}\mathrm{A}$. If the latter are all real, then it converges iff

$$0 < \alpha\lambda_i < 2 \qquad \forall i = 1, \ldots, n.$$

If both A and P are symmetric and positive definite matrices, the stationary Richardson method converges for any possible choice of $\mathbf{x}^{(0)}$ iff $0 < \alpha < 2/\lambda_{max}$, where $\lambda_{max}(> 0)$ is the maximum eigenvalues of $\mathrm{P}^{-1}\mathrm{A}$. Moreover, the spectral radius $\rho(\mathrm{B}_\alpha)$ of the iteration matrix $\mathrm{B}_\alpha = \mathrm{I} - \alpha\mathrm{P}^{-1}\mathrm{A}$ is minimized for $\alpha = \alpha_{opt}$, where

$$\boxed{\alpha_{opt} = \frac{2}{\lambda_{min} + \lambda_{max}}} \tag{5.58}$$

λ_{min} being the smallest eigenvalue of $\mathrm{P}^{-1}\mathrm{A}$. Finally, always when $\alpha = \alpha_{opt}$, the following convergence estimate holds

$$\boxed{\|\mathbf{e}^{(k)}\|_{\mathrm{A}} \leq \left(\frac{K(\mathrm{P}^{-1}\mathrm{A}) - 1}{K(\mathrm{P}^{-1}\mathrm{A}) + 1}\right)^k \|\mathbf{e}^{(0)}\|_{\mathrm{A}}, \quad k \geq 0} \tag{5.59}$$

where $\|\mathbf{v}\|_{\mathrm{A}} = \sqrt{\mathbf{v}^T\mathrm{A}\mathbf{v}}$, $\mathbf{v} \in \mathbb{R}^n$, is the so-called energy norm associated to the matrix A.

Notice that when A and P are symmetric positive definite matrices, then $\mathrm{P}^{-1}\mathrm{A}$ is similar to a symmetric positive definite matrix, then its eigenvalues are all real and positive (see Exercise 5.17).

Proposition 5.6 *If* $\mathrm{A} \in \mathbb{R}^{n\times n}$ *and* $\mathrm{P} \in \mathbb{R}^{n\times n}$ *are symmetric and positive definite matrices, the dynamic Richardson method converges if, for instance,* α_k *is chosen as follows:*

$$\boxed{\alpha_k = \frac{(\mathbf{z}^{(k)})^T\mathbf{r}^{(k)}}{(\mathbf{z}^{(k)})^T\mathrm{A}\mathbf{z}^{(k)}}, \qquad k \geq 0} \tag{5.60}$$

where $\mathbf{z}^{(k)} = \mathrm{P}^{-1}\mathbf{r}^{(k)}$ *is the preconditioned residual defined in* (5.54)*. With such choice for* α_k*, method* (5.53) *is called preconditioned gradient method or, simply, gradient method when* P *is the identity matrix.*
Finally the following convergence estimate holds

$$\boxed{\|\mathbf{e}^{(k)}\|_\mathrm{A} \leq \left(\frac{K(\mathrm{P}^{-1}\mathrm{A})-1}{K(\mathrm{P}^{-1}\mathrm{A})+1}\right)^k \|\mathbf{e}^{(0)}\|_\mathrm{A}, \quad k \geq 0} \tag{5.61}$$

The parameter α_k in (5.60) is the one that minimizes the new error $\|\mathbf{e}^{(k+1)}\|_\mathrm{A}$ (see Exercise 5.18).

In general, the dynamic version should be preferred to the stationary one since it does not require the knowledge of the extreme eigenvalues of $\mathrm{P}^{-1}\mathrm{A}$. As a matter of fact, the parameter α_k is determined in terms of quantities which are already available from the previous iteration.

We can rewrite the preconditioned gradient method more efficiently through the following algorithm (derivation is left as an exercise): given $\mathbf{x}^{(0)}$, set $\mathbf{r}^{(0)} = \mathbf{b} - \mathrm{A}\mathbf{x}^{(0)}$, then do

```
for $k = 0, 1, \ldots$
    $\mathrm{P}\mathbf{z}^{(k)} = \mathbf{r}^{(k)}$,
    $\alpha_k = \frac{(\mathbf{z}^{(k)})^T\mathbf{r}^{(k)}}{(\mathbf{z}^{(k)})^T\mathrm{A}\mathbf{z}^{(k)}}$,
    $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k\mathbf{z}^{(k)}$,
    $\mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} - \alpha_k\mathrm{A}\mathbf{z}^{(k)}$
```
(5.62)

The same algorithm can be used to implement the stationary Richardson method by simply replacing α_k with the constant value α.

From (5.59), we deduce that if $\mathrm{P}^{-1}\mathrm{A}$ is ill conditioned the convergence rate will be very low even for $\alpha = \alpha_{opt}$ (as in that case $\rho(\mathrm{B}_{\alpha_{opt}}) \simeq 1$). This is overcome by a suitable choice of P. This is the reason why P is called the preconditioner or the preconditioning matrix.

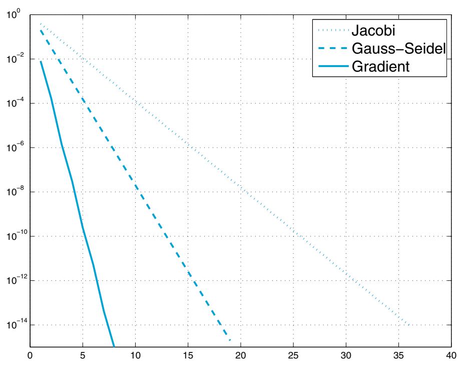


Figure 5.15. Convergence history for Jacobi, Gauss-Seidel and gradient methods applied to system (5.63)

If A is a generic matrix it may be a difficult task to find a preconditioner which guarantees an optimal trade-off between damping the condition number and keeping the computational cost for the solution of the system (5.54) reasonably low. The choice of P should be done taking into account the properties of the matrix A.

The dynamic Richardson method is implemented in Program 5.2 where the input parameter `P` stands for the preconditioning matrix (when not prescribed, the program implements the unpreconditioned method by setting P=I).

Example 5.14 This example, of theoretical interest only, has the purpose of comparing the convergence behavior of Jacobi, Gauss-Seidel and gradient methods applied to solve the following (mini) linear system:

$$2x_1 + x_2 = 1, \; x_1 + 3x_2 = 0 \tag{5.63}$$

with initial vector $\mathbf{x}^{(0)} = (1, 1/2)^T$. Note that the system matrix is symmetric and positive definite, and that the exact solution is $\mathbf{x} = (3/5, -1/5)^T$. We report in Figure 5.15 the behavior of the relative residual

$$E^{(k)} = \|\mathbf{r}^{(k)}\|/\|\mathbf{r}^{(0)}\| \tag{5.64}$$

for the three methods above. Iterations are stopped at the first iteration k_{min} for which $E^{(k_{min})} \leq 10^{-14}$. The gradient method appears to converge the fastest. ■

Example 5.15 Let us consider a system A$\mathbf{x} = \mathbf{b}$, where A $\in \mathbb{R}^{100\times 100}$ is a pentadiagonal matrix whose main diagonal has all entries equal to 4, while the first and third lower and upper diagonals have all entries equal to -1. As customary, $\mathbf{b}$ is chosen in such a way that $\mathbf{x} = (1, \ldots, 1)^T$ is the exact solution of our system. Let P be the tridiagonal matrix whose diagonal elements are all equal to 2, while the elements on the lower and upper diagonal are all equal to -1. Both A and P are symmetric and positive definite. With such a P as preconditioner, Program 5.2 can be used to implement the dynamic preconditioner Richardson method. We fix `tol=1.e-05`, `nmax=5000`, `x0=zeros(100,1)`. The method converges in 43 iterations. The same Program 5.2, used with `P='G'`, implements the Gauss-Seidel method; this time as many as 1658 iterations are required before satisfying the same stopping criterion. ■

5.11 The conjugate gradient method

In iterative schemes like (5.62) the new iterate $\mathbf{x}^{(k+1)}$ is obtained by adding to the old iterate $\mathbf{x}^{(k)}$ a vector, named *descent direction*, that is either the residual $\mathbf{r}^{(k)}$ or the preconditioned residual $\mathbf{z}^{(k)}$. A natural question is whether it is possible to find other descent directions, say $\mathbf{p}^{(k)}$, that ensure the convergence of the method in a lower number of iterations.

When the matrix A$\in \mathbb{R}^{n\times n}$ is symmetric and positive definite, the *conjugate gradient method* (in short, CG) makes use of a sequence of descent directions that are A*-orthogonal* (or A*-conjugate*), that is, $\forall k \geq 0$,

$$(\mathrm{A}\mathbf{p}^{(j)})^T\mathbf{p}^{(k+1)} = 0, \qquad j = 0, 1, \ldots, k. \tag{5.65}$$

For any vector $\mathbf{x}^{(0)}$, after setting $\mathbf{r}^{(0)} = \mathbf{b} - \mathrm{A}\mathbf{x}^{(0)}$ and $\mathbf{p}^{(0)} = \mathbf{r}^{(0)}$, the CG method takes the following form:

$$\boxed{\begin{aligned}
&\text{for } k = 0, 1, \ldots\\
&\quad \alpha_k = \frac{\mathbf{p}^{(k)^T}\mathbf{r}^{(k)}}{\mathbf{p}^{(k)^T}\mathrm{A}\mathbf{p}^{(k)}},\\
&\quad \mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k\mathbf{p}^{(k)},\\
&\quad \mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} - \alpha_k\mathrm{A}\mathbf{p}^{(k)},\\
&\quad \beta_k = \frac{(\mathrm{A}\mathbf{p}^{(k)})^T\mathbf{r}^{(k+1)}}{(\mathrm{A}\mathbf{p}^{(k)})^T\mathbf{p}^{(k)}},\\
&\quad \mathbf{p}^{(k+1)} = \mathbf{r}^{(k+1)} - \beta_k\mathbf{p}^{(k)}
\end{aligned}} \tag{5.66}$$

The parameter α_k guarantees that the error $\|\mathbf{e}^{(k+1)}\|_A$ is minimized along the descent direction $\mathbf{p}^{(k)}$, while β_k is chosen to ensure that the new direction $\mathbf{p}^{(k+1)}$ is A-conjugate with $\mathbf{p}^{(k)}$, that is $(\mathrm{A}\mathbf{p}^{(k)})^T\mathbf{p}^{(k+1)} = 0$. As a matter of fact, it can be proved (by the induction principle) that, if the latter relation is satisfied, then all orthogonality relations in (5.65) for $j = 0, \ldots, k-1$ are satisfied, too. For a complete derivation of the method, see for instance [QSS07, Chapter 4] or [Saa03]. It is possible to prove the following important result:

Proposition 5.7 *Let* A *be a symmetric and positive definite matrix. In exact arithmetic, the conjugate gradient method for solving (5.1) converges after at most* n *steps. Moreover, the error* $\mathbf{e}^{(k)}$ *at the* k*th iteration (with* $k < n$*) is orthogonal to* $\mathbf{p}^{(j)}$*, for* $j = 0, \dots, k-1$ *and*

$$\|\mathbf{e}^{(k)}\|_{\mathrm{A}} \leq \frac{2c^k}{1+c^{2k}} \|\mathbf{e}^{(0)}\|_{\mathrm{A}}, \tag{5.67}$$

with $c = \dfrac{\sqrt{K(\mathrm{A})}-1}{\sqrt{K(\mathrm{A})}+1}$.

Therefore, in absence of rounding errors, the CG method can be regarded as a direct method, since it terminates after a finite number of steps. However, for matrices of large size, it is usually employed as an iterative scheme, and the iterations are stopped when an error estimator (e.g. the relative residual (5.64)) falls below a fixed tolerance. In this respect, by comparing (5.67) with (5.61), it is readily seen that CG iterations converge more rapidly than gradient iterations, because of the presence of the square root of $K(\mathrm{A})$.

Also for the CG method it is possible to consider a preconditioned version (the PCG method), with a preconditioner P symmetric and positive definite, which reads as follows: given $\mathbf{x}^{(0)}$ and setting $\mathbf{r}^{(0)} = \mathbf{b} - \mathrm{A}\mathbf{x}^{(0)}$, $\mathbf{z}^{(0)} = \mathrm{P}^{-1}\mathbf{r}^{(0)}$ and $\mathbf{p}^{(0)} = \mathbf{z}^{(0)}$, do

```
for $k = 0, 1, \dots$
    $\alpha_k = \dfrac{{\mathbf{p}^{(k)}}^T \mathbf{r}^{(k)}}{{\mathbf{p}^{(k)}}^T \mathrm{A}\mathbf{p}^{(k)}}$,
    $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{p}^{(k)}$,
    $\mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} - \alpha_k \mathrm{A}\mathbf{p}^{(k)}$,                 (5.68)
    $\mathrm{P}\mathbf{z}^{(k+1)} = \mathbf{r}^{(k+1)}$,
    $\beta_k = \dfrac{(\mathrm{A}\mathbf{p}^{(k)})^T \mathbf{z}^{(k+1)}}{(\mathrm{A}\mathbf{p}^{(k)})^T \mathbf{p}^{(k)}}$,
    $\mathbf{p}^{(k+1)} = \mathbf{z}^{(k+1)} - \beta_k \mathbf{p}^{(k)}$
```

Table 5.4. Errors obtained using the preconditioned gradient method (PG), the preconditioned conjugate gradient method (PCG), and the direct method implemented in the MATLAB command \ for the solution of the Hilbert system. For the iterative methods also the number of iterations is reported

		\	PG		PCG	
n	$K(\mathrm{A}_n)$	Error	Error	Iter	Error	Iter
4	1.55e+04	7.72e-13	8.72e-03	995	1.12e-02	3
6	1.50e+07	7.61e-10	3.60e-03	1813	3.88e-03	4
8	1.53e+10	6.38e-07	6.30e-03	1089	7.53e-03	4
10	1.60e+13	5.24e-04	7.98e-03	875	2.21e-03	5
12	1.70e+16	6.27e-01	5.09e-03	1355	3.26e-03	5
14	6.06e+17	4.12e+01	3.91e-03	1379	4.32e-03	5

In this case the error estimate (5.64) still holds, however $K(\mathrm{A})$ is replaced by the more favourable $K(\mathrm{P}^{-1}\mathrm{A})$.
The PCG method is implemented in the MATLAB function `pcg`. `pcg`

Example 5.16 Let us go back to Example 5.9 on the Hilbert matrix and solve the related system (for different values of n) by the preconditioned gradient (PG) and the preconditioned conjugate gradient (PCG) methods, using as preconditioner the diagonal matrix D made of the diagonal entries of the Hilbert matrix. We fix $\mathbf{x}^{(0)}$ to be the null vector and iterate until the relative residual (5.64) is less than 10^{-6}. In Table 5.4 we report the absolute errors (with respect to the exact solution) obtained with PG and PCG methods, as well as the errors obtained using the MATLAB command \. For the latter, the error degenerates when n gets large. On the other hand, we can appreciate the beneficial effect that a suitable iterative method such as the PCG scheme can have on the number of iterations. ■

Remark 5.5 (Non-symmetric systems) The CG method is a special instance of the so-called *Krylov* (or *Lanczos*) *methods* that can be used for the solution of systems which are not necessarily symmetric. Their description is provided, e.g., in [Axe94], [Saa03] and [vdV03].

Some of them share with the CG method the notable property of finite termination, that is, in exact arithmetic they provide the exact solution in a finite number of iterations also for nonsymmetric systems. A remarkable example is the *GMRES* (Generalized Minimum RESidual) *method*, available in MATLAB under the name of `gmres`. `gmres`

Another method, the Bi-CGStab ([vdV03]), is very competitive with GMRES from the efficiency point of view. The MATLAB command is `bicgstab`. `bicgstab` ■

See Exercises 5.15-5.19.

5.12 When should an iterative method be stopped?

In theory, iterative methods require an infinite number of iterations to converge to the exact solution of a linear system. Even when this is not the case (see, e.g. the CG method), the number of iterations to achieve the solution within machine accuracy is very high when the size of the linear system gets large. In practice, aiming at the exact solution is neither reasonable nor necessary. Indeed, what we do really need is to obtain an approximation $\mathbf{x}^{(k)}$ for which we can guarantee that the error be lower than a desired tolerance ϵ. On the other hand, since the error is itself unknown (as it depends on the exact solution), we need a suitable *a posteriori* error estimator which predicts the error starting from quantities that have already been computed.

The first type of estimator is represented by the *residual* at the kth iteration, see (5.52). More precisely, we could stop our iterative method at the first iteration step k_{min} for which

$$\|\mathbf{r}^{(k_{min})}\| \leq \varepsilon \|\mathbf{b}\|.$$

Setting $\widehat{\mathbf{x}} = \mathbf{x}^{(k_{min})}$ and $\mathbf{r} = \mathbf{r}^{(k_{min})}$ in (5.36) we would obtain

$$\frac{\|\mathbf{e}^{(k_{min})}\|}{\|\mathbf{x}\|} \leq \varepsilon K(\mathrm{A}),$$

which is an estimate for the relative error. We deduce that the control on the residual is meaningful only for those matrices whose condition number is reasonably small.

Example 5.17 Let us consider the linear system (5.1) where A=A$_{20}$ is the Hilbert matrix of dimension 20 introduced in Example 5.9 and $\mathbf{b}$ is constructed in such a way that the exact solution is $\mathbf{x} = (1, 1, \dots, 1)^T$. Since A is symmetric and positive definite the Gauss-Seidel method surely converges. We use Program 5.2 to solve this system taking `x0` to be the null initial vector and setting a tolerance on the residual equal to 10^{-5}. The method converges in 472 iterations; however the relative error is very large and equals 0.0586. This is due to the fact that A is extremely ill conditioned, having $K(\mathrm{A}) \simeq 10^{17}$. In Figure 5.16 we show the behavior of the residual (normalized to the initial one) and that of the error as the number of iterations increases. ■

An alternative approach is based on the use of a different error estimator, namely the *increment* $\boldsymbol{\delta}^{(k)} = \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}$. More precisely, we can stop our iterative method at the first iteration step k_{min} for which

$$\|\boldsymbol{\delta}^{(k_{min})}\| \leq \varepsilon. \tag{5.69}$$

In the special case where B is symmetric and positive definite, we have

$$\|\mathbf{e}^{(k)}\| = \|\mathbf{e}^{(k+1)} + \boldsymbol{\delta}^{(k)}\| \leq \rho(\mathrm{B})\|\mathbf{e}^{(k)}\| + \|\boldsymbol{\delta}^{(k)}\|.$$

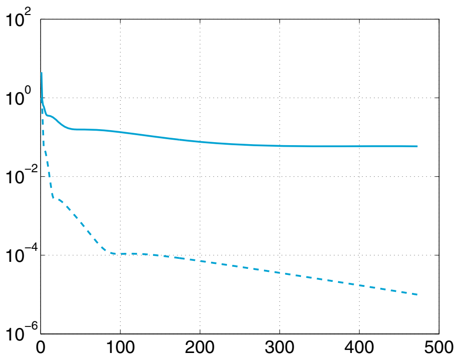


Figure 5.16. Behavior, versus iterations k, of the relative residual (5.64) (*dashed line*) and of the error $\|\mathbf{x}-\mathbf{x}^{(k)}\|/\|\mathbf{x}\|$ (*solid line*) for Gauss-Seidel iterations applied to the system of Example 5.17

Since $\rho(\mathrm{B})$ should be less than 1 in order for the method to converge, we deduce

$$\boxed{\|\mathbf{e}^{(k)}\| \leq \frac{1}{1-\rho(\mathrm{B})}\|\boldsymbol{\delta}^{(k)}\|} \tag{5.70}$$

From the last inequality we see that the control on the increment is meaningful only if $\rho(\mathrm{B})$ is much smaller than 1 since in that case the error will be of the same size as the increment.

In fact, the same conclusion holds even if B is not symmetric and positive definite (as it occurs for the Jacobi and Gauss-Seidel methods); however in that case (5.70) is no longer true.
Should one be interested in relative errors, (5.69) could be replaced by

$$\frac{\|\boldsymbol{\delta}^{(k_{min})}\|}{\|\mathbf{b}\|} \leq \varepsilon$$

and, consequently, (5.70) by

$$\frac{\|\mathbf{e}^{(k)}\|}{\|\mathbf{b}\|} \leq \frac{1}{1-\rho(\mathrm{B})}\varepsilon.$$

Example 5.18 Let us consider a system whose matrix A$\in\mathbb{R}^{50\times 50}$ is tridiagonal and symmetric with entries equal to 2.001 on the main diagonal and equal to 1 on the two other diagonals. As usual, the right hand side $\mathbf{b}$ is chosen in such a way that the unit vector $(1,\ldots,1)^T$ is the exact solution. Since A is tridiagonal with strict diagonal dominance, the Gauss-Seidel method will converge about twice as fast as the Jacobi method (in view of Proposition 5.4). Let us use Program 5.2 to solve our system in which we replace the stopping criterion based on the residual by that based on the increment, i.e. $\|\boldsymbol{\delta}^{(k)}\| \leq \varepsilon$. Using the initial vector whose components are $(\mathbf{x}_0)_i = 10\sin(100i)$ (for $i=1,\ldots,n$)

and setting the tolerance `tol`$= 10^{-5}$, after 859 iterations the solution returned by the program is such that $\|\mathbf{e}^{(859)}\| \simeq 0.0021$. The convergence is very slow and the error is quite large since the spectral radius of the iteration matrix is equal to 0.9952, which is very close to 1. Should the diagonal entries be set equal to 3, after only 17 iterations we would have obtained convergence with an error $\|\mathbf{e}^{(17)}\| \simeq 8.96 \cdot 10^{-6}$. In fact in that case the spectral radius of the iteration matrix would be equal to 0.443. ■

Let us summarize

1. An iterative method for the solution of a linear system starts from a given initial vector $\mathbf{x}^{(0)}$ and builds up a sequence of vectors $\mathbf{x}^{(k)}$ which we require to converge to the exact solution as $k \to \infty$;
2. an iterative method converges for every possible choice of the initial vector $\mathbf{x}^{(0)}$ iff the spectral radius of the iteration matrix is strictly less than 1;
3. classical iterative methods are those of Jacobi and Gauss-Seidel. A sufficient condition for convergence is that the system matrix be strictly diagonally dominant by row (or symmetric and definite positive in the case of Gauss-Seidel);
4. in the Richardson method convergence is accelerated thanks to the introduction of a parameter and (possibly) a convenient preconditioning matrix;
5. with the conjugate gradient method the exact solution of a symmetric positive definite system can be computed in a finite number of iterations (in exact arithmetic). This method can be generalized to the nonsymmetric case;
6. there are two possible stopping criteria for an iterative method: controlling the residual or controlling the increment. The former is meaningful if the system matrix is well conditioned, the latter if the spectral radius of the iteration matrix is not close to 1.

5.13 To wrap-up: direct or iterative?

In this section we compare direct and iterative methods on several simple test cases. For a linear system of small size, it doesn't really matter since every method will make the job. Instead, for large scale systems, the choice will depend primarily on the matrix properties (such as symmetry, positive definiteness, sparsity pattern, condition number), but also on the kind of available computer resources (memory access, fast processors, etc.). We must admit that in our tests the comparison will not be fully loyal. One direct solver that we will in fact use is the MATLAB built-in function \ which is compiled and optimized, whereas the iterative solvers are not. Our computations were carried out on a processor Intel® Core™2 Duo 2.53GHz with 3072KB cache and 3GByte RAM.

A sparse, banded linear system with small bandwidth
The first test case concerns linear systems arising from the 5-point finite difference discretizations of the Poisson problem on the square $(-1,1)^2$ with homogeneous Dirichlet boundary conditions (see Section 9.2.4). Uniform grids of step $h = 2/(N+1)$ in both spatial coordinates are considered, for several values of N. The corresponding finite difference matrices, with $(N+2)^2$ rows and columns, are generated using Program 9.2. On Figure 5.17, left, we plot the matrix structure corresponding to the value $(N+2)^2 = 256$ (obtained by the command `spy`): it is sparse, banded, with only 5 non-null entries per row. After eliminating those rows and columns associated to boundary nodes, we denote by $n = N^2$ the size of the reduced matrix. Any such matrix is symmetric and positive definite but ill conditioned: its spectral condition number behaves like a constant time h^{-2} for all values of h, that is the smaller the parameter h, the worse the matrix condition number. To solve the associated linear systems we will use the Cholesky factorization, the preconditioned conjugate gradient method (PCG) with preconditioner given by the incomplete Cholesky factorization, and the MATLAB command \ that, in the current case, is in fact an ad hoc algorithm for pentadiagonal symmetric matrices. The incomplete Cholesky factorization of A is generated from an algebraic manipulation of the entries of the R factor of A (see [QSS07]) and is computed by the command `ichol(A,struct('type','ict','droptol',1e-03))`.

spy

ichol

The stopping criterion for the PCG method is that the relative residual (5.64) be lower than 10^{-13}; the CPU time is also inclusive of the time necessary to construct the preconditioner.

In Figure 5.17, right, we compare the CPU time for the three different methods versus the matrix size. The direct method hidden by the command \ is by far the cheapest: in fact, it is based on a variant of the Gaussian elimination that is particularly effective for sparse banded matrices with small bandwith.

The PCG method, in its turn, is more convenient than the CG method (with no preconditioning). For instance, if $n = 4096$ (corresponding to $N = 64$) the PCG method requires 18 iterations, whereas the CG method would require 154 iterations. Both methods, however, are less convenient than the Cholesky factorization. We warn the reader that the conclusions should be taken with a grain of salt, as they depend on the way the algorithms are implemented and the kind of computer used.

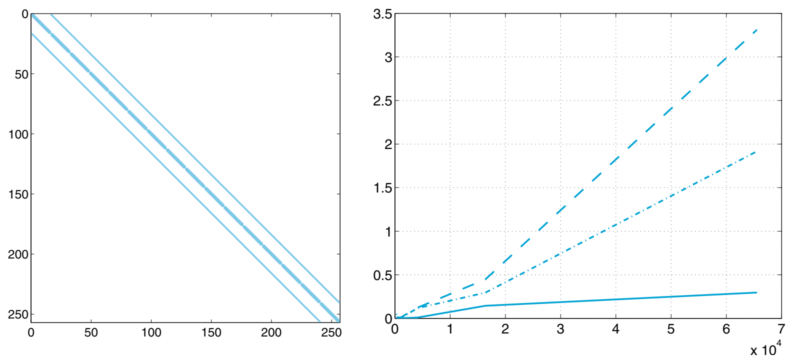


Figure 5.17. The structure of the matrix for the first test case (*left*), and the CPU time (in sec.) needed for the solution of the associated linear system (*right*): the *solid line* refers to the command \, the *dashed-dotted line* to the use of the Cholesky factorization, the *dashed line* to the PCG iterative method. The values in abscissa refer to the matrix dimension n

The case of a broad band

We still consider the same Poisson equation, however this time the discretization is based on spectral methods with Gauss-Legendre-Lobatto quadrature formulae (see, for instance, [Qua13, CHQZ06]). Even though the number of grid-nodes is the same as for the finite differences, with spectral methods the derivatives are approximated using many more nodes (in fact, at any given node the x-derivatives are approximated using all the nodes sitting on the same row, whereas all those on the same column are used to compute y-derivatives). The corresponding matrices are still sparse and structured, however the number of non-null entries is definitely higher than in the former case. This is clear from the example in Figure 5.18, left, where the spectral matrix has still $N^2 = 256$ rows and columns, but the number of nonzero entries is 7936 instead of the 1216 of the finite difference matrix of Figure 5.17.

The CPU time reported in Figure 5.18, right, shows that for this matrix the PCG algorithm, using the incomplete Cholesky factorization as preconditioner, performs much better than the other two methods.

A first conclusion to draw is that for sparse symmetric and positive definite matrices with large bandwidth, PCG is more efficient than the direct method implemented in MATLAB (which does not use the Cholesky factorization since the matrix is stored with the format `sparse`). We point out that a suitable preconditioner is however crucial in order for the PCG method to become competitive.

Finally, we shoud keep in mind that direct methods require more memory storage than iterative methods, a difficulty that could become insurmontable in large scale applications.

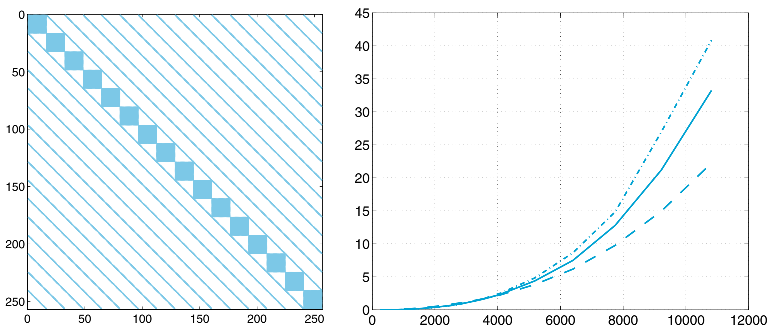


Figure 5.18. The structure of the matrix used in the second test case (*left*), and the CPU time (in sec.) needed to solve the associated linear system (*right*): the *solid line* refers to the command \, the *dashed-dotted line* to the use of the Cholesky factorization, the *dashed line* to the PCG iterative method. The values in abscissa refer to the matrix dimension n

Systems with full matrices

With the MATLAB command `gallery` we can get access to a collection of matrices featuring different structure and properties. In particular for our third test case, by the command `A=gallery('riemann',n)` we select the so-called Riemannn matrix of dimension `n`, that is a `n` $\times$ `n` full, non-symmetric matrix whose determinant behaves like $\det(\mathtt{A}) = \mathcal{O}(\mathtt{n}!\mathtt{n}^{-1/2+\epsilon})$ for all $\epsilon > 0$. The associated linear system is solved by the iterative GMRES method (see Remark 5.5) and the iterations will be stopped as soon as the norm of the relative residual (5.64) becomes less than 10^{-13}. Alternatively, we will use the MATLAB command \ that, in the case at hand, implements the LU factorization.

`gallery`

For several values of n we will solve the corresponding linear system whose exact solution is the unitary vector: the right-hand side is computed accordingly. The GMRES iterations are obtained without preconditioning. In Figure 5.19, right, we report the CPU time for n ranging between 100 and 1000. On the left we report `cond(A)`, the condition number of A. As we can see, the direct factorization method is far less expensive than the un-preconditioned GMRES method, however it becomes more expensive for large n when suitable preconditioners are used.

Octave 5.1 The `gallery` command is not available in Octave. However a few are available such as the Hilbert, Hankel or Vandermonde matrices, see the commands `hankel`, `hilb`, `invhilb` `sylvester_matrix`, `toeplitz` and `vander`. Moreover if you have access to MATLAB, you can save a matrix defined in the gallery using the `save` command and then load it in Octave using `load`. Here is an example:
In MATLAB:

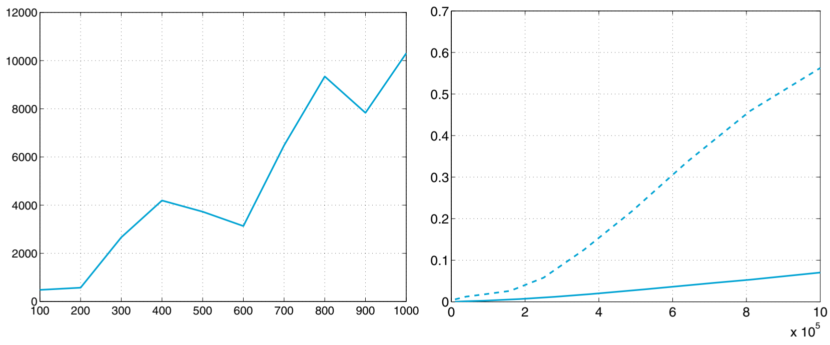


Figure 5.19. On the left, the condition number of the Riemann matrix A. On the right, the comparison between the CPU times (in sec.) for the solution of the linear system: the *solid line* refers to the command \, the *dashed line* refers to the GMRES iterative method with no preconditioning. The values in abscissa refer to the matrix dimension n

```
riemann10=gallery('riemann',10);
save 'riemann10' riemann10
```

In Octave:

```
load 'riemann10' riemann10
```

■

Systems with sparse, nonsymmetric matrices

We consider linear systems that are generated by the finite element discretization of diffusion-transport-reaction boundary-value problems in two dimensions. These problems are similar to the one reported in (9.17) which refers to a one-dimensional case. Its finite element approximation, that is illustrated in Section 9.2.3 in the one-dimensional case, makes use of piecewise linear polynomials to represent the solution in each triangular element of a grid that partitions the region where the boundary-value problem is set up. The unknowns of the associated algebraic system is the set of values attained by the solution at the vertices of the internal triangles. We refer to, e.g., [QV94] for a description of this method, as well as for the determination of the entries of the matrix. Let us simply point out that this matrix is sparse, but not banded (its sparsity pattern depends on the way the vertices are numbered) and nonsymmetric, due to the presence of the transport term. The lack of symmetry, however, is not evident from the representation of its structure in Figure 5.20, left.

The smaller the *diameter* h of the triangles (i.e. the lengths of their longest edge), the higher the matrix size. We are using unstructured triangular grids generated by the MATLAB toolbox `pdetool`. We have compared the CPU time necessary to solve the linear system correspond-

pdetool

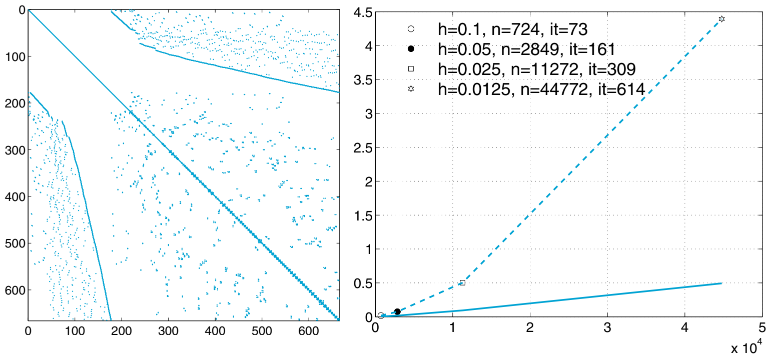


Figure 5.20. The structure of one of the matrices used for the fourth test case (*left*), and the CPU time (in sec.) needed for the solution of the associated linear system (*right*): the *solid line* refers to the command \, the *dashed line* to the Bi-CGStab iterative method. The values in abscissa refer to the matrix dimension n, while *it* stands for Bi-CGStab iterations

ing to the case $h = 0.1, 0.05, 0.025$, and 0.0125. We have used the MATLAB command \, that in this case uses the UMFPACK library and the (MATLAB implementation of the) iterative method Bi-CGStab which can be regarded as a generalization to nonsymmetric systems of the conjugate gradient method. In abscissae we have reported the number of unknowns that ranges from 724 (for $h = 0.1$) to 44772 (for $h = 0.0125$). Also in this case, the direct method is less expensive than the iterative one. Should we use as preconditioner for the Bi-CGStab method the incomplete LU factorization, the number of iterations would reduce, however the CPU time would be higher than the one for the unpreconditioned case. The incomplete LU factorization of the matrix A is generated from an algebraic manipulation of the entries of the factors L and U of A (see, e.g., [QSS07]) and is computed by the command `ilu(A,struct('type','ilutp','droptol',1.e-3))`. ilu

In conclusion

The comparisons that we have carried out, although very limited, outlines a few relevant aspects. In general, direct methods (especially if implemented in their most sophisticated versions, such as in the \ MATLAB command) are more efficient than iterative methods when the latter are used without efficient preconditioners. However, they are more sensitive to the matrix ill conditioning (see for instance the Example 5.16) and may require a substantial amount of storage.

A further aspect that is worth mentioning is that direct methods require the knowledge of the matrix entries, whereas iterative methods don't. In fact, what is nedeed at each iteration is the computation of matrix-vector products for given vectors. This aspect makes iterative methods especially interesting for those problems in which the matrix is not explicitely generated.

5.14 What we haven't told you

Several efficient variants of the LU factorization are available for sparse systems of large dimension. Among the most advanced, we quote the so-called *multifrontal method* which makes use of a suitable reordering of the system unknowns in order to keep the triangular factors L and U as sparse as possible. The multifrontal method is implemented in the software package UMFPACK. More on this issue is available on [GL96] and [DD99].

Concerning iterative methods, both the conjugate gradient method and the GMRES method are special instances of Krylov methods. For a description of Krylov methods see e.g. [Axe94], [Saa03] and [vdV03].

As it was pointed out, iterative methods converge slowly if the system matrix is severely ill conditioned. Several preconditioning strategies have been developed (see, e.g., [dV89] and [vdV03]). Some of them are purely algebraic, that is, they are based on incomplete (or inexact) factorizations of the given system matrix, and are implemented in the already quoted MATLAB functions `ichol` and `ilu`. Other strategies are developed *ad hoc* by exploiting the physical origin and the structure of the problem which has generated the linear system at hand.

Finally it is worthwhile to mention the *multigrid methods* which are based on the sequential use of a hierarchy of systems of variable dimensions that "resemble" the original one, allowing a clever error reduction strategy (see, e.g., [Hac85], [Wes04] and [Hac94]).

Octave 5.2 In Octave, `ichol` is not yet available. Only the incomplete LU factorization has been implemented in the function `luinc`. See the help of Octave. ■

5.15 Exercises

Exercise 5.1 For a given matrix $A \in \mathbb{R}^{n\times n}$ find the number of operations (as a function of n) that are needed for computing its determinant by the recursive formula (1.8).

`magic` **Exercise 5.2** Use the MATLAB command `magic(n)`, $n = 3, 4, \ldots, 500$, to construct the magic squares of order n, that is, those matrices having entries for which the sum of the elements by rows, columns or diagonals are identical. Then compute their determinants by the command `det` introduced in Section

1.4 and the CPU time that is needed for this computation using the `cputime` command. Finally, approximate this data by the least-squares method and deduce that the CPU time scales approximately as n^3.

Exercise 5.3 Find for which values of ε the matrix defined in (5.16) does not satisfy the hypotheses of Proposition 5.1. For which value of ε does this matrix become singular? Is it possible to compute the LU factorization in that case?

Exercise 5.4 Verify that the number of operations necessary to compute the LU factorization of a square matrix A of dimension n is approximately $2n^3/3$.

Exercise 5.5 Show that the LU factorization of A can be used for computing the inverse matrix A^{-1}. (Observe that the jth column vector of A^{-1}, say $\mathbf{x}_j$, satisfies the linear system $\mathrm{A}\mathbf{x}_j = \mathbf{e}_j$, $\mathbf{e}_j$ being the vector whose components are all null except the jth component which is 1.)

Exercise 5.6 Compute the factors L and U of the matrix of Example 5.8 and verify that the LU factorization is inaccurate.

Exercise 5.7 Explain why partial pivoting by row is not convenient for symmetric matrices.

Exercise 5.8 Consider the linear system $\mathrm{A}\mathbf{x} = \mathbf{b}$ with

$$\mathrm{A} = \begin{bmatrix} 2 & -2 & 0 \\ \varepsilon - 2 & 2 & 0 \\ 0 & -1 & 3 \end{bmatrix},$$

and $\mathbf{b}$ such that the corresponding solution is $\mathbf{x} = (1, 1, 1)^T$ and ε is a positive real number. Compute the LU factorization of A and note that $l_{32} \to \infty$ when $\varepsilon \to 0$. Verify that the computed solution is not affected by rounding errors when $\varepsilon = 10^{-k}$ with $k = 0, .., 9$ and $\mathbf{b} = (0, \varepsilon, 2)^T$. Moreover, analyze the relative error on the exact solution when $\varepsilon = 1/3 \cdot 10^{-k}$ with $k = 0, .., 9$, and the exact solution is $\mathbf{x}_{ex} = (\log(5/2), 1, 1)^T$.

Exercise 5.9 Consider the linear systems $\mathrm{A}_i\mathbf{x}_i = \mathbf{b}_i$, $i = 1, 2, 3$, with

$$\mathrm{A}_1 = \begin{bmatrix} 15 & 6 & 8 & 11 \\ 6 & 6 & 5 & 3 \\ 8 & 5 & 7 & 6 \\ 11 & 3 & 6 & 9 \end{bmatrix}, \ \mathrm{A}_i = (\mathrm{A}_1)^i, \ i = 2, 3,$$

and $\mathbf{b}_i$ such that the solution is always $\mathbf{x}_i = (1, 1, 1, 1)^T$. Solve the system by the LU factorization using partial pivoting by row, and comment on the obtained results.

Exercise 5.10 Show that for a symmetric and positive definite matrix A we have $K(\mathrm{A}^2) = (K(\mathrm{A}))^2$.

Exercise 5.11 Analyse the convergence properties of the Jacobi and Gauss-Seidel methods for the solution of a linear system whose matrix is

$$\mathrm{A} = \begin{bmatrix} \alpha & 0 & 1 \\ 0 & \alpha & 0 \\ 1 & 0 & \alpha \end{bmatrix}, \qquad \alpha \in \mathbb{R}.$$

Exercise 5.12 Provide a sufficient condition on β so that both the Jacobi and Gauss-Seidel methods converge when applied for the solution of a system whose matrix is

$$\mathrm{A} = \begin{bmatrix} -10 & 2 \\ \beta & 5 \end{bmatrix}. \tag{5.71}$$

Exercise 5.13 For the solution of the linear system $\mathrm{A}\mathbf{x} = \mathbf{b}$ with $\mathrm{A} \in \mathbb{R}^{n\times n}$, consider the *relaxation method*: given $\mathbf{x}^{(0)} = (x_1^{(0)}, \ldots, x_n^{(0)})^T$, for $k = 0, 1, \ldots$ compute

$$r_i^{(k)} = b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)},\ x_i^{(k+1)} = (1-\omega)x_i^{(k)} + \omega\frac{r_i^{(k)}}{a_{ii}},$$

for $i = 1, \ldots, n$, where ω is a real parameter. Find the explicit form of the corresponding iterative matrix, then verify that the condition $0 < \omega < 2$ is necessary for the convergence of this method. Note that if $\omega = 1$ this method reduces to the Gauss-Seidel method. If $1 < \omega < 2$ the method is known as *SOR* (*successive over-relaxation*).

Exercise 5.14 Consider the linear system $\mathrm{A}\mathbf{x} = \mathbf{b}$ with $\mathrm{A} = \begin{bmatrix} 3 & 2 \\ 2 & 6 \end{bmatrix}$ and say whether the Gauss-Seidel method converges, without explicitly computing the spectral radius of the iteration matrix. Repeat with $\mathrm{A} = \begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix}$.

Exercise 5.15 Compute the first iteration of the Jacobi, Gauss-Seidel and preconditioned gradient method (with preconditioner given by the diagonal of A) for the solution of system (5.63) with $\mathbf{x}^{(0)} = (1, 1/2)^T$.

Exercise 5.16 Prove (5.58), then show that

$$\rho(\mathrm{B}_{\alpha_{opt}}) = \frac{\lambda_{max} - \lambda_{min}}{\lambda_{max} + \lambda_{min}} = \frac{K(\mathrm{P}^{-1}\mathrm{A}) - 1}{K(\mathrm{P}^{-1}\mathrm{A}) + 1}. \tag{5.72}$$

Exercise 5.17 Prove that if A and P are symmetric positive definite matrices, then $\mathrm{P}^{-1}\mathrm{A}$ is similar to a symmetric positive definite matrix.

Exercise 5.18 Note that, in using an acceleration parameter α instead of α_k, from (5.62) we have $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha\mathbf{z}^{(k)}$ so that the error $\mathbf{e}^{(k+1)} = \mathbf{x} - \mathbf{x}^{(k+1)}$ depends on α. Prove that the expression of α_k given in (5.60) minimizes the function $\Phi(\alpha) = \|\mathbf{e}^{(k+1)}\|_{\mathrm{A}}^2$ with respect to $\alpha \in \mathbb{R}$.

Exercise 5.19 Let us consider a set of $n = 20$ factories which produce 20 different goods. With reference to the Leontief model introduced in Problem 5.3, suppose that the matrix C has the following integer entries: $c_{ij} = i + j$ for $i, j = 1, \dots, n$, while $b_i = i$, for $i = 1, \dots, 20$. Is it possible to solve this system by the gradient method? Propose a method based on the gradient method noting that, if A is nonsingular, the matrix $\mathrm{A}^T\mathrm{A}$ is symmetric and positive definite.

6

Eigenvalues and eigenvectors

Given a square matrix $\mathrm{A} \in \mathbb{C}^{n\times n}$, the eigenvalue problem consists in finding a scalar λ (real or complex) and a nonnull vector $\mathbf{x}$ such that

$$\boxed{\mathrm{A}\mathbf{x} = \lambda\mathbf{x}} \tag{6.1}$$

Any such λ is called an *eigenvalue* of A, while $\mathbf{x}$ is the associated *eigenvector*. The latter is not unique; indeed all its multiples $\alpha\mathbf{x}$ with $\alpha \neq 0$, real or complex, are also eigenvectors associated with λ. Should $\mathbf{x}$ be known, λ can be recovered by using the *Rayleigh quotient* $\mathbf{x}^H\mathrm{A}\mathbf{x}/\|\mathbf{x}\|^2$, $\mathbf{x}^H = \bar{\mathbf{x}}^T$ being the vector whose ith component is equal to $\bar{x}_i$.

A number λ is an eigenvalue of A if it is a root of the following polynomial of degree n (called the *characteristic polynomial* of A):

$$p_{\mathrm{A}}(\lambda) = \det(\mathrm{A} - \lambda\mathrm{I}).$$

Consequently, a square matrix of dimension n has exactly n eigenvalues (real or complex), not necessarily distinct. Also, if A has real entries, $p_{\mathrm{A}}(\lambda)$ has real coefficients, and therefore complex eigenvalues of A necessarily occur in complex conjugate pairs.

Let us also recall that a matrix $\mathrm{A}\in \mathbb{C}^{n\times n}$ is said to be diagonalizable if there exists a nonsingular matrix $\mathrm{U}\in \mathbb{C}^{n\times n}$ such that

$$\mathrm{U}^{-1}\mathrm{A}\mathrm{U} = \Lambda = \mathrm{diag}(\lambda_1, \ldots, \lambda_n). \tag{6.2}$$

The columns of U are the eigenvectors of A and form a basis for $\mathbb{C}^n$.

In the special case where A is either diagonal or triangular, its eigenvalues are nothing but its diagonal entries. However, if A is a general matrix and its dimension n is sufficiently large, seeking the zeros of $p_{\mathrm{A}}(\lambda)$ is not the most convenient approach. Ad hoc algorithms are better suited, and some of them will be described in the next sections.

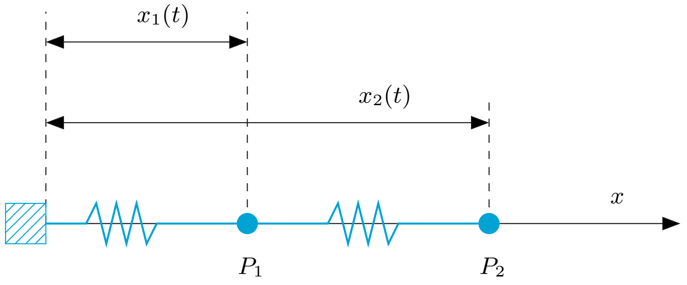


Figure 6.1. The system of two pointwise bodies of equal mass connected by springs

6.1 Some representative problems

Problem 6.1 (Elastic springs) Consider the system of Figure 6.1 made of two pointwise bodies P_1 and P_2 of mass m, connected by two springs and free to move along the line joining P_1 and P_2. Let $x_i(t)$ denote the position occupied by P_i at time t for $i = 1, 2$. Then from the second law of dynamics we obtain

$$m\,\ddot{x}_1 = K(x_2 - x_1) - Kx_1, \qquad m\,\ddot{x}_2 = K(x_1 - x_2),$$

where K is the elasticity coefficient of both springs. We are interested in free oscillations whose corresponding solution is $x_i = a_i \sin(\omega t + \phi)$, $i = 1, 2$, with $a_i \neq 0$. In this case we find that

$$-\,ma_1\omega^2 = K(a_2 - a_1) - Ka_1, \qquad -ma_2\omega^2 = K(a_1 - a_2). \quad (6.3)$$

This is a 2×2 homogeneous system which has a non-trivial solution $\mathbf{a} = (a_1, a_2)^T$ iff the number $\lambda = m\omega^2/K$ is an eigenvalue of the matrix

$$\mathrm{A} = \begin{bmatrix} 2 & -1 \\ -1 & 1 \end{bmatrix}.$$

With this definition of λ, (6.3) becomes $\mathrm{A}\mathbf{a} = \lambda\mathbf{a}$. Since $p_{\mathrm{A}}(\lambda) = (2 - \lambda)(1 - \lambda) - 1$, the two eigenvalues are $\lambda_1 \simeq 2.618$ and $\lambda_2 \simeq 0.382$ and correspond to the frequencies of oscillation $\omega_i = \sqrt{K\lambda_i/m}$ which are admitted by our system. ■

Problem 6.2 (Population dynamics) Several mathematical models have been proposed in order to predict the evolution of certain species (either human or animal). The simplest population model, which was introduced in 1920 by Lotka and formalized by Leslie 20 years later, is based on the rate of mortality and fecundity for different age intervals, say $i = 0, \ldots, n$. Let $x_i^{(t)}$ denote the number of females (males don't matter in this context) whose age at time t falls in the ith interval. The values of $x_i^{(0)}$ are given. Moreover, let s_i denote the rate of survival of the females belonging to the ith interval, and m_i the average number of females generated from a female in the ith interval.

The model by Lotka and Leslie is described by the set of equations

$$\begin{aligned} x_{i+1}^{(t+1)} &= x_i^{(t)} s_i \qquad i = 0, \ldots, n-1, \\ x_0^{(t+1)} &= \sum_{i=0}^{n} x_i^{(t)} m_i. \end{aligned}$$

The n first equations describe the population development, the last its reproduction. In matrix form we have

$$\mathbf{x}^{(t+1)} = \mathrm{A}\mathbf{x}^{(t)},$$

where $\mathbf{x}^{(t)} = (x_0^{(t)}, \ldots, x_n^{(t)})^T$ while A is the *Leslie matrix*

$$\mathrm{A} = \begin{bmatrix} m_0 & m_1 & \ldots\ldots & & m_n \\ s_0 & 0 & \ldots\ldots & & 0 \\ 0 & s_1 & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & s_{n-1} & 0 \end{bmatrix}.$$

We will see in Section 6.2 that the dynamics of this population is determined by the eigenvalue of maximum modulus of A, say λ_1, whereas the distribution of the individuals in the different age intervals (normalized with respect to the whole population), is obtained as the limit of $\mathbf{x}^{(t)}$ for $t \to \infty$ and satisfies $\mathrm{A}\mathbf{x} = \lambda_1 \mathbf{x}$. This problem will be solved in Exercise 6.2. ■

Problem 6.3 (Interurban railway network) For n given cities, let A be the matrix whose entry a_{ij} is equal to 1 if the ith city is directly connected to the jth city, and 0 otherwise. One can show that the components of the eigenvector $\mathbf{x}$ (of unit length) associated with the maximum eigenvalue provides the accessibility rate (which is a measure of the ease of access) to the various cities. In Example 6.2 we will compute this vector for the case of the railways system of the eleven most important cities in Lombardy (see Figure 6.2). ■

Problem 6.4 (Image compression) The problem of image compression can be faced using the singular-value decomposition of a matrix introduced in (5.45). Indeed, a black and white image can be represented by a real $m \times n$ rectangular matrix A where m and n represent

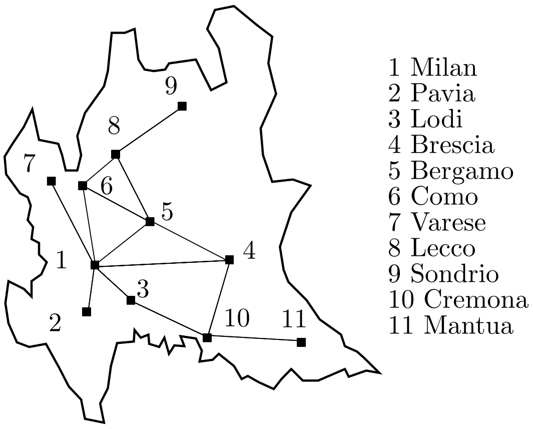


Figure 6.2. A schematic representation of the railway network between the main cities of Lombardy

the number of *pixels* that are present in the horizontal and vertical direction, respectively, and the coefficient a_{ij} represents the intensity of gray of the (i,j)th pixel. Considering the singular value decomposition (5.45) of A, and denoting by $\mathbf{u}_i$ and $\mathbf{v}_i$ the ith column vectors of U and V, respectively, we find

$$\mathrm{A} = \sigma_1 \mathbf{u}_1 \mathbf{v}_1^T + \sigma_2 \mathbf{u}_2 \mathbf{v}_2^T + \ldots + \sigma_p \mathbf{u}_p \mathbf{v}_p^T. \tag{6.4}$$

We can approximate A by the matrix A_k which is obtained by truncating the sum (6.4) to the first k terms, for $1 \leq k \leq p$. If the singular values σ_i are in decreasing order, $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_p$, disregarding the latter $p-k$ should not significantly affect the quality of the image. To transfer the "compressed" image A_k (for instance from one computer to another) we simply need to transfer the vectors $\mathbf{u}_i$, $\mathbf{v}_i$ and the singular values σ_i for $i = 1, \ldots, k$ and not all the entries of A. In Example 6.9 we will see this technique in action. ■

Even though most of the methods that we will present in this Section are valid for compex matrices too, for simplicity we will limit our analysis to real matrices. In any case, we note that MATLAB and Octave programs for computing both eigenvalues and eigenvectors work on both real and complex variables, with no need to modify the calling instructions.

6.2 The power method

As noticed in Problems 6.2 and 6.3, the knowledge of the whole *spectrum* of A (that is the set of all its eigenvalues) is not always required. Often, only the *extremal* eigenvalues matter, that is, those having largest and smallest modulus.

Suppose that A is a square matrix of dimension n, with real entries, and assume that its eigenvalues are ordered as follows

$$|\lambda_1| > |\lambda_2| \geq |\lambda_3| \geq \ldots \geq |\lambda_n|. \tag{6.5}$$

Note, in particular, that $|\lambda_1|$ is distinct from the other moduli of the eigenvalues of A. Let us indicate by $\mathbf{x}_1$ the eigenvector (with unit length) associated with λ_1. If the eigenvectors of A are linearly independent, λ_1 and $\mathbf{x}_1$ can be computed by the following iterative procedure, commonly known as the *power method*:

given an arbitrary initial vector $\mathbf{x}^{(0)} \in \mathbb{C}^n$ and setting $\mathbf{y}^{(0)} = \mathbf{x}^{(0)}/\|\mathbf{x}^{(0)}\|$, compute

$$\boxed{\begin{array}{l}\text{for } k = 1, 2, \ldots \\[2ex] \mathbf{x}^{(k)} = \mathrm{A}\mathbf{y}^{(k-1)}, \quad \mathbf{y}^{(k)} = \dfrac{\mathbf{x}^{(k)}}{\|\mathbf{x}^{(k)}\|}, \quad \lambda^{(k)} = (\mathbf{y}^{(k)})^H \mathrm{A}\mathbf{y}^{(k)}\end{array}} \tag{6.6}$$

Note that, by recursion, one finds $\mathbf{y}^{(k)} = \beta^{(k)}\mathrm{A}^k\mathbf{x}^{(0)}$ where $\beta^{(k)} = (\prod_{i=0}^{k}\|\mathbf{x}^{(i)}\|)^{-1}$ for $k \geq 1$. The presence of the powers of A justifies the name given to this method.

In the next section we will see that this method generates a sequence of vectors $\{\mathbf{y}^{(k)}\}$ with unit length which, as $k \to \infty$, align themselves along the direction of the eigenvector $\mathbf{x}_1$.

It is possible to prove (see, e.g. [QSS07]) that, if ${\mathbf{x}^{(0)}}^H\mathbf{x}_1 \neq 0$, both quantities $\|\mathbf{y}^{(k)} - ({\mathbf{y}^{(k)}}^H\mathbf{x}_1)\mathbf{x}_1\|$ and $|\lambda^{(k)} - \lambda_1|$ are proportional to the ratio $|\lambda_2/\lambda_1|^k$ in the case of a generic matrix, and to $|\lambda_2/\lambda_1|^{2k}$ when the matrix A is hermitian. In all cases $\lambda^{(k)} \to \lambda_1$ for $k \to \infty$.

An implementation of the power method is given in the Program 6.1. The iterative procedure is stopped at the first iteration k when

$$|\lambda^{(k)} - \lambda^{(k-1)}| < \varepsilon|\lambda^{(k)}|,$$

where ε is a desired tolerance. The input parameters are the real matrix `A`, the tolerance `tol` for the stopping test, the maximum admissible number of iterations `nmax` and the initial vector `x0`. Output parameters are the maximum modulus eigenvalue `lambda`, the associated eigenvector and the actual number of iterations which have been carried out.

Program 6.1. eigpower: power method

```
function [lambda,x,iter]=eigpower(A,tol,nmax,x0)
%EIGPOWER Computes the eigenvalue with maximum modulus
%   of a real matrix.
%   LAMBDA=EIGPOWER(A) computes with the power method
%   the eigenvalue of A of maximum modulus from an
%   initial guess which by default is an all one vector.
%   LAMBDA=EIGPOWER(A,TOL,NMAX,X0) uses an absolute
%   error tolerance TOL (the default is 1.e-6) and a
%   maximum number of iterations NMAX (the default is
%   100), starting from the initial vector X0.
%   [LAMBDA,V,ITER]=EIGPOWER(A,TOL,NMAX,X0) also returns
%   the eigenvector V such that A*V=LAMBDA*V and the
%   iteration number at which V was computed.
[n,m] = size(A);
if n ~= m, error('Only for square matrices'); end
if nargin == 1
   tol = 1.e-06;   x0 = ones(n,1);   nmax = 100;
end
x0 = x0/norm(x0);
pro = A*x0;
lambda = x0'*pro;
err = tol*abs(lambda) + 1;
iter = 0;
while err>tol*abs(lambda) & abs(lambda)~=0 & iter<=nmax
   x = pro;                  x = x/norm(x);
   pro = A*x;                lambdanew = x'*pro;
   err = abs(lambdanew - lambda);
   lambda = lambdanew;   iter = iter + 1;
end
return
```

Example 6.1 Consider the family of matrices

$$A(\alpha) = \begin{bmatrix} \alpha & 2 & 3 & 13 \\ 5 & 11 & 10 & 8 \\ 9 & 7 & 6 & 12 \\ 4 & 14 & 15 & 1 \end{bmatrix}, \qquad \alpha \in \mathbb{R}.$$

We want to approximate the eigenvalue with largest modulus by the power method. When $\alpha = 30$, the eigenvalues of the matrix are given by $\lambda_1 = 39.396$, $\lambda_2 = 17.8208$, $\lambda_3 = -9.5022$ and $\lambda_4 = 0.2854$ (only the first four significant digits are reported). The method approximates λ_1 in 22 iterations with a tolerance $\varepsilon = 10^{-10}$ and $\mathbf{x}^{(0)} = \mathbf{1}^T$. However, if $\alpha = -30$ we need as many as 708 iterations. The different behavior can be explained by noting that in the latter case one has $\lambda_1 = -30.643$, $\lambda_2 = 29.7359$, $\lambda_3 = -11.6806$ and $\lambda_4 = 0.5878$. Thus, $|\lambda_2|/|\lambda_1| = 0.9704$ is close to unity. ■

Example 6.2 (Interurban railway network) We denote by A$\in \mathbb{R}^{11\times 11}$ the matrix associated to the railways system of Figure 6.2, i.e. the matrix whose entry a_{ij} is equal to one if there is a direct connection between the ith and the jth cities, zero otherwise. Setting `tol=1.e-12` and `x0=ones(11,1)`, after 26 iterations Program 6.1 returns the following approximation of the eigenvector (of unitary length) associated to the eigenvalue of maximum modulus of A:

```
x' =
  Columns 1 through 8
  0.5271  0.1590  0.2165  0.3580  0.4690  0.3861  0.1590  0.2837
  Columns 9 through 11
  0.0856  0.1906  0.0575
```

The most reachable city is Milan, which is the one associated to the first component of x (the largest in modulus), the least one is Mantua, which is associated to the last component of x, that of minimum modulus. Of course our analysis accounts solely for the existence of connections among the cities but not on how frequent these connections are. ■

6.2.1 Convergence analysis

Since we have assumed that the eigenvectors $\mathbf{x}_1, \ldots, \mathbf{x}_n$ of A are linearly independent, they form a basis for $\mathbb{C}^n$. By expanding $\mathbf{x}^{(0)}$ and $\mathbf{y}^{(0)}$ as follows

$$\mathbf{x}^{(0)} = \sum_{i=1}^{n} \alpha_i \mathbf{x}_i,\ \mathbf{y}^{(0)} = \beta^{(0)} \sum_{i=1}^{n} \alpha_i \mathbf{x}_i, \quad \text{with } \beta^{(0)} = 1/\|\mathbf{x}^{(0)}\| \text{ and } \alpha_i \in \mathbb{C},$$

at the first step the power method yields

$$\mathbf{x}^{(1)} = \mathrm{A}\mathbf{y}^{(0)} = \beta^{(0)} \mathrm{A} \sum_{i=1}^{n} \alpha_i \mathbf{x}_i = \beta^{(0)} \sum_{i=1}^{n} \alpha_i \lambda_i \mathbf{x}_i$$

and, similarly,

$$\mathbf{y}^{(1)} = \beta^{(1)} \sum_{i=1}^{n} \alpha_i \lambda_i \mathbf{x}_i, \qquad \beta^{(1)} = \frac{1}{\|\mathbf{x}^{(0)}\| \ \|\mathbf{x}^{(1)}\|}.$$

At a given step k we will have

$$\mathbf{y}^{(k)} = \beta^{(k)} \sum_{i=1}^{n} \alpha_i \lambda_i^k \mathbf{x}_i, \qquad \beta^{(k)} = \frac{1}{\|\mathbf{x}^{(0)}\| \cdots \|\mathbf{x}^{(k)}\|}$$

and therefore

$$\mathbf{y}^{(k)} = \lambda_1^k \beta^{(k)} \left(\alpha_1 \mathbf{x}_1 + \sum_{i=2}^{n} \alpha_i \frac{\lambda_i^k}{\lambda_1^k} \mathbf{x}_i \right).$$

Since $|\lambda_i/\lambda_1| < 1$ for $i = 2, \ldots, n$, the vector $\mathbf{y}^{(k)}$ tends to align along the same direction as the eigenvector $\mathbf{x}_1$ when k tends to $+\infty$, provided $\alpha_1 \neq 0$.

The previous argument would not apply if either the values $|\lambda_1^k \beta^{(k)} \alpha_1|$ diverge to $+\infty$ when $|\lambda_1| > 1$ or converge to 0 when $|\lambda_1| < 1$. Fortunately, this can not happen; actually, we can prove that $|\lambda_1^k \beta^{(k)} \alpha_1| \to 1$ when $k \to \infty$. As a matter of fact, by an induction argument we have

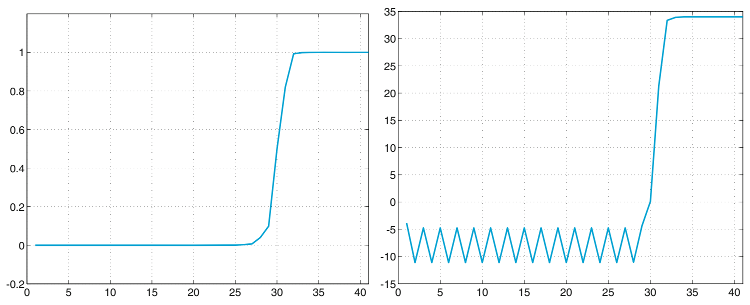


Figure 6.3. The values ${\mathbf{y}^{(k)}}^H\mathbf{x}_1$ (*at left*) and $\lambda^{(k)}$ (*at right*) of Example 6.3, when $k = 1, \ldots, 41$

$$\beta^{(k)} = \frac{1}{\|\mathrm{A}^k\mathbf{x}^{(0)}\|} = \frac{1}{\|\lambda_1^k(\alpha_1\mathbf{x}_1 + \mathbf{r}^{(k)})\|} \qquad \text{for } k \geq 1,$$

where $\mathbf{r}^{(k)} = \sum_{i=2}^{n} \alpha_i(\lambda_i/\lambda_1)^k\mathbf{x}_i$ and, since $\mathbf{r}^{(k)} \to \mathbf{0}$ for $k \to \infty$ and $\|\mathbf{x}_1\| = 1$, it holds

$$|\lambda_1^k\beta^{(k)}\alpha_1| = \frac{|\lambda_1^k\alpha_1|}{\|\lambda_1^k(\alpha_1\mathbf{x}_1 + \mathbf{r}^{(k)})\|} \to 1.$$

Finally, we have ${\mathbf{y}^{(k)}}^H\mathbf{x}_1 = \lambda_1^k\beta^{(k)}\alpha_1$, so that the values $\|\mathbf{y}^{(k)} - ({\mathbf{y}^{(k)}}^H\mathbf{x}_1)\mathbf{x}_1\|$ behave like the dominant term in the vector $\mathbf{r}^{(k)}$, that is they converge to zero like $|\lambda_2/\lambda_1|^k$ for $k \to \infty$.

The condition on α_1, which is impossible to ensure in practice since $\mathbf{x}_1$ is unknown, is in fact not restrictive. Actually, the effect of roundoff errors is the appearance of a non-null component along the direction of $\mathbf{x}_1$, even though this was not the case for the initial vector $\mathbf{x}^{(0)}$. (We can say that this is one of the rare circumstances where roundoff errors help us!)

Example 6.3 Consider the matrix $\mathrm{A}(\alpha)$ of Example 6.1, with $\alpha = 16$. The eigenvector $\mathbf{x}_1$ of unit length associated with λ_1 is $(1/2, 1/2, 1/2, 1/2)^T$. Let us choose (on purpose!) the initial vector $(2, -2, 3, -3)^T$, which is orthogonal to $\mathbf{x}_1$. We report in Figure 6.3 the quantity $\cos(\theta^{(k)}) = (\mathbf{y}^{(k)})^H\mathbf{x}_1$ versus k. We can see that after about 30 iterations of the power method the cosine tends to 1 and the angle $\theta^{(k)}$ tends to 0, while the sequence $\lambda^{(k)}$ approaches $\lambda_1 = 34$. The power method has therefore generated, thanks to the roundoff errors, a sequence of vectors $\mathbf{y}^{(k)}$ whose component along the direction of $\mathbf{x}_1$ is increasingly relevant. ■

It is possible to prove that the power method converges even if λ_1 is a multiple root of $p_\mathrm{A}(\lambda)$. On the contrary it does not converge when there exist two distinct eigenvalues both with maximum modulus. In that case the sequence $\lambda^{(k)}$ does not converge to any limit, rather it oscillates between two values.

See Exercises 6.1-6.3.

6.3 Generalization of the power method

A first possible generalization of the power method consists in applying it to the inverse of the matrix A (provided A is non singular!). Since the eigenvalues of A^{-1} are the reciprocals of those of A, the power method in that case allows us to approximate the eigenvalue of A of minimum modulus. In this way we obtain the so-called *inverse power method*:

given an initial vector $\mathbf{x}^{(0)}$, we set $\mathbf{y}^{(0)} = \mathbf{x}^{(0)}/\|\mathbf{x}^{(0)}\|$ and compute

$$\boxed{\begin{array}{l}\text{for } k=1,2,\dots\\[1em] \mathbf{x}^{(k)} = \mathrm{A}^{-1}\mathbf{y}^{(k-1)},\ \mathbf{y}^{(k)} = \dfrac{\mathbf{x}^{(k)}}{\|\mathbf{x}^{(k)}\|},\ \mu^{(k)} = (\mathbf{y}^{(k)})^H\mathrm{A}^{-1}\mathbf{y}^{(k)}\end{array}} \qquad (6.7)$$

If A admits n linearly independent eigenvectors, and if also the eigenvalue λ_n of minimum modulus is distinct from the others, then

$$\lim_{k\to\infty}\mu^{(k)} = 1/\lambda_n,$$

i.e. $(\mu^{(k)})^{-1}$ tends to λ_n for $k\to\infty$.

At each step k we have to solve a linear system of the form $\mathrm{A}\mathbf{x}^{(k)} = \mathbf{y}^{(k-1)}$. It is therefore convenient to generate the LU factorization of A (or its Cholesky factorization if A is symmetric and positive definite) once for all, and then solve two triangular systems at each iteration.

It is worth noticing that the `lu` command (in MATLAB and in Octave) can generate the LU decomposition even for complex matrices.

A further generalization of the power method is useful to approximate the (unknown) eigenvalue of A nearest to a given number μ (either real or complex). Let λ_μ denote such eigenvalue and let us define the shifted matrix $\mathrm{A}_\mu = \mathrm{A} - \mu\mathrm{I}$, whose eigenvalues are $\lambda(\mathrm{A}_\mu) = \lambda(\mathrm{A}) - \mu$. In order to approximate λ_μ, we can at first approximate the eigenvalue of minimum length of A_μ, say $\lambda_{min}(\mathrm{A}_\mu)$, by applying the inverse power method to A_μ, and then compute $\lambda_\mu = \lambda_{min}(\mathrm{A}_\mu) + \mu$. This technique is known as the *power method with shift*, and the number μ is called the *shift*.

In Program 6.2 we implement the inverse power method with shift. The inverse power method (without shift) is recovered by simply setting $\mu = 0$.

The input parameter `mu` is the shift, while the other parameters are as in Program 6.1. Output parameters are the approximation of the eigenvalue λ_μ of A, its associated eigenvector `x` and the actual number of iterations that have been carried out.

Program 6.2. invshift: inverse power method with shift

```
function [lambda,x,iter]=invshift(A,mu,tol,nmax,x0)
%INVSHIFT Inverse power method with shift
%  LAMBDA=INVSHIFT(A) computes the  eigenvalue of A of
%  minimum modulus with the inverse power method.
%  LAMBDA=INVSHIFT(A,MU) computes the eigenvalue of A
%  closest to the given number (real or complex) MU.
%  LAMBDA=INVSHIFT(A,MU,TOL,NMAX,X0) uses an absolute
%  error tolerance TOL (the default is 1.e-6) and a
%  maximum number of iterations NMAX (the default is
%  100), starting from the initial vector X0.
%  [LAMBDA,V,ITER]=INVSHIFT(A,MU,TOL,NMAX,X0) also
%  returns the eigenvector V such that A*V=LAMBDA*V and
%  the iteration number at which V was computed.
[n,m]=size(A);
if n ~= m, error('Only for square matrices'); end
if nargin == 1
  x0 = rand(n,1); nmax = 100; tol = 1.e-06; mu = 0;
elseif nargin == 2
  x0 = rand(n,1); nmax = 100; tol = 1.e-06;
end
[L,U]=lu(A-mu*eye(n));
if norm(x0) == 0
   x0 = rand(n,1);
end
x0=x0/norm(x0);
z0=L\x0;
pro=U\z0;
lambda=x0'*pro;
err=tol*abs(lambda)+1;        iter=0;
while err>tol*abs(lambda)&abs(lambda)~=0&iter<=nmax
   x = pro; x = x/norm(x);
   z=L\x;    pro=U\z;
   lambdanew = x'*pro;
   err = abs(lambdanew - lambda);
   lambda = lambdanew;
   iter = iter + 1;
end
lambda = 1/lambda + mu;
return
```

Example 6.4 Let us apply the inverse power method to compute the minimum modulus eigenvalue of the matrix A(30) defined in Example 6.1. Program 6.2, called by the instruction

```
[lambda,x,iter]=invshift(A(30))
```

converges in 5 iterations to the value 0.2854. ■

Example 6.5 For the matrix A(30) of Example 6.1 we seek the eigenvalue closest to the value 17. For that we use Program 6.2 with `mu=17`, `tol`

=10^{-10} and `x0=[1;1;1;1]`. After 8 iterations the Program returns the value `lambda=17.82079703055703`. A less accurate knowledge of the *shift* would involve more iterations. For instance, if we set `mu=13` the program returns the value 17.82079703064106 after 19 iterations. ■

The value of the shift can be modified during the iterations, by setting $\mu = \lambda^{(k)}$. This yields a faster convergence; however the computational cost grows substantially since now at each iteration the matrix A_μ does change and the LU factorization has to be performed at each iteration.

See Exercises 6.4-6.6.

6.4 How to compute the shift

In order to successfully apply the power method with shift we need to locate (more or less accurately) the eigenvalues of A in the complex plane. To this end let us introduce the following definition.

Let A be a square matrix of dimension n. The *Gershgorin circles* $C_i^{(r)}$ and $C_i^{(c)}$ associated with its ith row and ith column are respectively defined as

$$C_i^{(r)} = \{z \in \mathbb{C} : \ |z - a_{ii}| \leq \sum_{j=1, j\neq i}^{n} |a_{ij}|\},$$
$$C_i^{(c)} = \{z \in \mathbb{C} : \ |z - a_{ii}| \leq \sum_{j=1, j\neq i}^{n} |a_{ji}|\}.$$

$C_i^{(r)}$ is called the ith *row circle* and $C_i^{(c)}$ the ith *column circle.*

By the Program 6.3 we can visualize in two different windows (that are opened by the command `figure`) the row circles and the column circles of a matrix. The command `hold on` allows the overlapping of subsequent pictures (in our case, the different circles that have been computed in sequential mode). This command can be neutralized by the command `hold off`. The commands `title`, `xlabel` and `ylabel` have the aim of visualizing the title and the axis labels in the figure.

`figure`
`hold on`
`hold off`
`title`
`xlabel`
`ylabel`

The command `patch` was used in order to color the circles, while the command `axis image` sets scaling for the x- and y-axes on the current plot.

`patch`
`axis`

Program 6.3. gershcircles: Gershgorin circles

```
function gershcircles(A)
%GERSHCIRCLES plots the Gershgorin circles
%  GERSHCIRCLES(A) draws the Gershgorin circles for
%   the square matrix A and its transpose.
n = size(A);
```

```
if n(1) ~= n(2)
  error('Only square matrices');
else
  n = n(1); circler = zeros(n,201); circlec = circler;
end
center = diag(A);
radiic = sum(abs(A-diag(center)));
radiir = sum(abs(A'-diag(center)));
one = ones(1,201); cosisin = exp(i*[0:pi/100:2*pi]);
figure(1); title('Row circles','Fontsize',18);
set(gca,'Fontsize',18)
xlabel('Re'); ylabel('Im');
figure(2); title('Column circles','Fontsize',18);
set(gca,'Fontsize',18)
xlabel('Re'); ylabel('Im');
color=[0.8,1,1];
for k = 1:n
  circlec(k,:) = center(k)*one + radiic(k)*cosisin;
  circler(k,:) = center(k)*one + radiir(k)*cosisin;
  figure(1);
  patch(real(circler(k,:)),imag(circler(k,:)),...
  color(1,:));
  hold on
  plot(real(circler(k,:)),imag(circler(k,:)),'k-',...
     real(center(k)),imag(center(k)),'kx');
  figure(2);
  patch(real(circlec(k,:)),imag(circlec(k,:)),...
  color(1,:));
  hold on
  plot(real(circlec(k,:)),imag(circlec(k,:)),'k-',...
     real(center(k)),imag(center(k)),'kx');
end
for k = 1:n
  figure(1);
  plot(real(circler(k,:)),imag(circler(k,:)),'k-',...
     real(center(k)),imag(center(k)),'kx');
  figure(2);
  plot(real(circlec(k,:)),imag(circlec(k,:)),'k-',...
     real(center(k)),imag(center(k)),'kx');
end
return
```

Example 6.6 In Figure 6.4 we have plotted the Gershgorin circles associated with the matrix

$$\mathrm{A} = \begin{bmatrix} 30 & 1 & 2 & 3 \\ 4 & 15 & -4 & -2 \\ -1 & 0 & 3 & 5 \\ -3 & 5 & 0 & -1 \end{bmatrix}.$$

The centers of the circles have been identified by a cross. ■

As previously anticipated, Gershgorin circles may be used to locate the eigenvalues of a matrix, as stated in the following proposition.

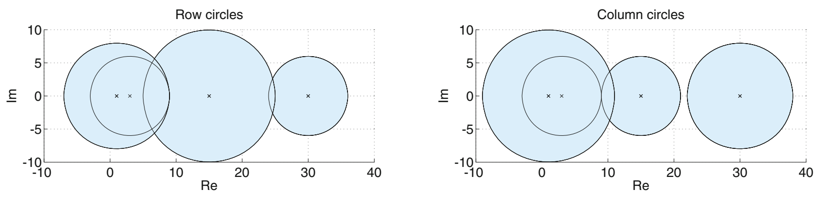


Figure 6.4. Row circles (*left*) and column circles (*right*) for the matrix of Example 6.6

Proposition 6.1 *All the eigenvalues of a given matrix A*$\in \mathbb{C}^{n\times n}$ *belong to the region of the complex plane which is the intersection of the two regions formed respectively by the union of the row circles and column circles.*
Moreover, should m *row circles (or column circles), with* $1 \leq m \leq n$*, be disconnected from the union of the remaining* $n-m$ *circles, then their union contains exactly* m *eigenvalues.*

There is no guarantee that a circle should contain eigenvalues, unless it is isolated from the others. The information provided by Ghersghorin circles are in general quite coarse, thus the previous result can provide only a preliminary guess of the shift.

Note that from Proposition 6.1, all the eigenvalues of a strictly diagonally dominant matrix are non-null.

Example 6.7 From the analysis of the row circles of the matrix A(30) of Example 6.1 we deduce that the real parts of the eigenvalues of A lie between -32 and 48. Thus we can use Program 6.2 to compute the maximum modulus eigenvalue by setting the value of the shift μ equal to 48. The convergence is achieved in 15 iterations, whereas 22 iterations would be required using the power method with the same initial guess `x0=[1;1;1;1]` and the same tolerance `tol=1.e-10`. ■

Let us summarize

1. The power method is an iterative procedure to compute the eigenvalue of maximum modulus of a given matrix;
2. the inverse power method allows the computation of the eigenvalue of minimum modulus; to efficiently implement the method, it is advisible to factorize the given matrix before iterations start;
3. the power method with shift allows the computation of the eigenvalue closest to a given number; its effective application requires some *a-*

priori knowledge of the location of the eigenvalues of the matrix, which can be achieved by inspecting the Gershgorin circles.

See Exercises 6.7-6.8.

6.5 Computation of all the eigenvalues

Two square matrices A and B having the same dimension are called *similar* if there exists a non singular matrix P such that

$$\mathrm{P}^{-1}\mathrm{AP} = \mathrm{B}.$$

Similar matrices share the same eigenvalues. Indeed, if λ is an eigenvalue of A and $\mathbf{x} \neq \mathbf{0}$ is an associated eigenvector, we have

$$\mathrm{BP}^{-1}\mathbf{x} = \mathrm{P}^{-1}\mathrm{A}\mathbf{x} = \lambda \mathrm{P}^{-1}\mathbf{x},$$

that is, λ is also an eigenvalue of B and its associated eigenvector is now $\mathbf{y} = \mathrm{P}^{-1}\mathbf{x}$.

The methods which allow a simultaneous approximation of all the eigenvalues of a matrix are generally based on the idea of transforming A (after an infinite number of steps) into a similar matrix with either diagonal or triangular form, whose eigenvalues are therefore given by the entries lying on its main diagonal.

Among these methods we mention the *QR method* which is implemented in MATLAB in the function `eig`. More precisely, the command `D=eig(A)` returns a vector `D` containing all the eigenvalues of `A`. However, by setting `[X,D]=eig(A)`, we obtain two matrices: the diagonal matrix `D` formed by the eigenvalues of `A`, and a matrix `X` whose column vectors are the eigenvectors of `A`. Thus, `A*X=X*D`.

`eig`

The method of QR iterations is called in this way since it makes a repeated use of the QR factorization introduced in Section 5.7 to compute the eigenvalues of the matrix A. Here we present the QR method only for real matrices and in its most elementary form (whose convergence is not always guaranteed). For a more complete description of this method we refer to [QSS07, Chapter 5], whereas for its extension to the complex case we refer to [GL96, Section 5.2.10] and [Dem97, Section 4.2.1].

The idea consists in building a sequence of matrices $\mathrm{A}^{(k)}$, each of them similar to A. After setting $\mathrm{A}^{(0)} = \mathrm{A}$, at each $k = 0, 1, \ldots$, using the QR factorization we compute the square matrices $\mathrm{Q}^{(k+1)}$ and $\mathrm{R}^{(k+1)}$ such that

$$\mathrm{Q}^{(k+1)}\mathrm{R}^{(k+1)} = \mathrm{A}^{(k)},$$

whence we set $\mathrm{A}^{(k+1)} = \mathrm{R}^{(k+1)}\mathrm{Q}^{(k+1)}$.

The matrices $\mathrm{A}^{(k)}$, $k = 0, 1, 2, \ldots$ are all similar, thus they share with A their eigenvalues (see Exercise 6.9). Moreover, if $\mathrm{A} \in \mathbb{R}^{n\times n}$ and its eigenvalues satisfy $|\lambda_1| > |\lambda_2| > \ldots > |\lambda_n|$, then

$$\lim_{k\to+\infty} \mathrm{A}^{(k)} = \mathrm{T} = \begin{bmatrix} \lambda_1 & t_{12} & \ldots & t_{1n} \\ 0 & \ddots & \ddots & \vdots \\ \vdots & & \lambda_{n-1} & t_{n-1,n} \\ 0 & \ldots & 0 & \lambda_n \end{bmatrix}. \tag{6.8}$$

The rate of decay to zero of the lower triangular coefficients, $a_{i,j}^{(k)}$ for $i > j$, when k tends to infinity, depends on $\max_i |\lambda_{i+1}/\lambda_i|$. In practice, the iterations are stopped when $\max_{i>j} |a_{i,j}^{(k)}| \leq \epsilon$, $\epsilon > 0$ being a given tolerance.

Under the further assumption that A is symmetric, the sequence $\{\mathrm{A}^{(k)}\}$ converges to a diagonal matrix.

Program 6.4 implements the QR iteration method. The input parameters are the matrix `A`, the tolerance `tol` and the maximum number of iterations allowed, `nmax`.

Program 6.4. qrbasic: method of QR iterations

```
function D=qrbasic(A,tol,nmax)
%QRBASIC computes all the eigenvalues of a matrix A.
%  D=QRBASIC(A,TOL,NMAX) computes by QR iterations all
%  the eigenvalues of A within a tolerance TOL and a
%  maximum number of iteration NMAX. The convergence of
%  this method is not always guaranteed.
[n,m]=size(A);
if n ~= m, error('The matrix must be squared'); end
T = A; niter = 0;  test = max(max(abs(tril(A,-1))));
while niter <= nmax & test > tol
    [Q,R]=qr(T);    T = R*Q;
    niter = niter + 1;
    test = max(max(abs(tril(T,-1))));
end
if niter > nmax
  warning(['The method does not converge '...
           'in the maximum number of iterations\n']);
else
  fprintf(['The method converges in ' ...
           '%i iterations\n'],niter);
end
D = diag(T);
return
```

Example 6.8 Let us consider the matrix A(30) of Example 6.1 and call Program 6.4 to compute its eigenvalues. We obtain

```
D=qrbasic(A(30),1.e-14,100)
```

```
The method converges in 56 iterations
D =
   39.3960
   17.8208
   -9.5022
    0.2854
```

These eigenvalues are in good agreement with those reported in Example 6.1, that were obtained with the command **eig**. The convergence rate decreases when there are eigenvalues whose moduli are almost the same. This is the case of the matrix corresponding to $\alpha = -30$: two eigenvalues have about the same modulus and the method requires as many as 1149 iterations to converge within the same tolerance

```
D=qrbasic(A(-30),1.e-14,2000)

The method converges in 1149 iterations
D =
  -30.6430
   29.7359
  -11.6806
    0.5878
```

■

A special case is the one of large sparse matrices. In this case, if **A** is stored in a sparse mode the command `eigs(A,k)` allows the computation of the k first eigenvalues of **A** having larger modulus.

eigs

Example 6.9 (Image compression) With the MATLAB command `A=imread('lena'.'jpg')` we upload a black and white JPEG image. (This is indeed a very popular image as it is commonly used by the scientific community to test programs for image compression.) The variable **A** is a matrix of 512 by 512 eight-bit integer numbers (**uint8**) that represent the intensity of gray. By the commands

imread

```
image(A); colormap(gray(256));
```

we obtain the first image on the left hand of Figure 6.5. To compute the SVD of A we must first convert A in a double precision matrix (the floating-point numbers usually used by MATLAB), and then invoke the commands

```
A=double(A); [U,S,V]=svd(A);
```

In the middle of Figure 6.5 we report the image that is obtained by using only the first 20 singular values of **S**, through the commands

```
k=20; X=U(:,1:k)*S(1:k,1:k)*(V(:,1:k))';
image(uint8(X));colormap(gray(256));
```

The third image on the right-hand of Figure 6.5 is obtained using the first 60 singular values. It requires the storage of 61500 coefficients (two matrices of 512×60 entries plus the first 60 singular values) instead of 262144 coefficients of the original image. ■

Octave 6.1 The syntax of the **imread** command in Octave reads

```
imread('lena.jpg')
```

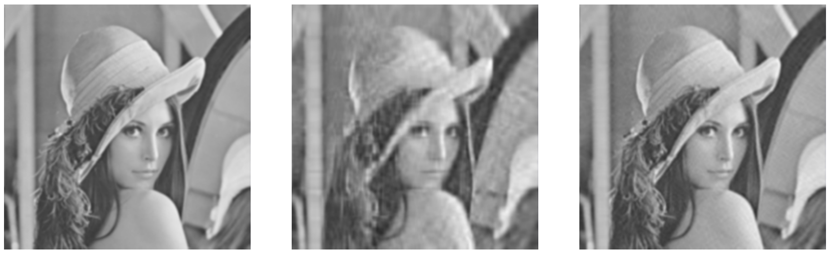


Figure 6.5. The original image (*left*) and those obtained using the first 20 (*center*) and 60 (*right*) singular values, respectively

Note that it slightly differs from that of MATLAB. ■

Let us summarize

1. The method of QR iterations allows the approximation of all the eigenvalues of a given matrix A;
2. in its basic version, this method is guaranteed to converge if A has real coefficients and distinct eigenvalues;
3. its asymptotic rate of convergence depends on the largest modulus of the ratio of two successive eigenvalues.

See Exercises 6.9-6.10.

6.6 What we haven't told you

We have not analyzed the issue of the condition number of the eigenvalue problem, which measures the sensitivity of the eigenvalues to the variation of the entries of the matrix. The interested reader is advised to refer to, for instance, [Wil88], [GL96] and [QSS07, Chapter 5].

Let us just remark that the eigenvalue computation is not necessarily an ill conditioned problem when the condition number of the matrix is large. An instance of this is provided by the Hilbert matrix (see Example 5.10): although its condition number is extremely large, the eigenvalue computation of the Hilbert matrix is well conditioned thanks to the fact that the matrix is symmetric and positive definite.

Besides the QR method, for computing simultaneously all the eigenvalues we can use the Jacobi method which transforms a symmetric matrix into a diagonal matrix, by eliminating, step-by-step, through similarity transformations, every off-diagonal element. This method does not terminate in a finite number of steps since, while a new off-diagonal element is set to zero, those previously treated can reassume non-zero values.

Other methods are the Lanczos method and the method which uses the so-called Sturm sequences. For a survey of all these methods see [Saa92].

arpackc

The MATLAB library ARPACK (available through the command `arpackc`) can be used to compute the eigenvalues of large matrices. The MATLAB function `eigs` is a command that uses this library.

Let us mention that an appropriate use of the *deflation* technique (which consists in a successive elimination of the eigenvalues already computed) allows the acceleration of the convergence of the previous methods and hence the reduction of their computational cost.

6.7 Exercises

Exercise 6.1 Upon setting the tolerance equal to $\varepsilon = 10^{-10}$, use the power method to approximate the maximum modulus eigenvalue for the following matrices, starting from the initial vector $\mathbf{x}^{(0)} = (1,2,3)^T$:

$$\mathrm{A}_1 = \begin{bmatrix} 1 & 2 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}, \; \mathrm{A}_2 = \begin{bmatrix} 0.1 & 3.8 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}, \; \mathrm{A}_3 = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}.$$

Then comment on the convergence behavior of the method in the three different cases.

Exercise 6.2 (Population dynamics) The features of a population of fishes are described by the following Leslie matrix introduced in Problem 6.2:

i	Age interval (months)	$x_i^{(0)}$	m_i	s_i
0	0-3	6	0	0.2
1	3-6	12	0.5	0.4
2	6-9	8	0.8	0.8
3	9-12	4	0.3	–

Find the vector $\mathbf{x}$ of the normalized distribution of this population for different age intervals, according to what we have seen in Problem 6.2.

Exercise 6.3 Prove that the power method does not converge for matrices featuring an eigenvalue of maximum modulus $\lambda_1 = \gamma e^{i\vartheta}$ and another eigenvalue $\lambda_2 = \gamma e^{-i\vartheta}$, where $i = \sqrt{-1}$, $\gamma \in \mathbb{R} \setminus \{0\}$ and $\vartheta \in \mathbb{R} \setminus \{k\pi,\ k \in \mathbb{Z}\}$.

Exercise 6.4 Show that the eigenvalues of A^{-1} are the reciprocals of those of A.

Exercise 6.5 Verify that the power method is unable to compute the maximum modulus eigenvalue of the following matrix, and explain why:

$$\mathrm{A} = \begin{bmatrix} \frac{1}{3} & \frac{2}{3} & 2 & 3 \\ 1 & 0 & -1 & 2 \\ 0 & 0 & -\frac{5}{3} & -\frac{2}{3} \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

Exercise 6.6 By using the power method with shift, compute the largest positive eigenvalue and the negative eigenvalue of largest modulus of

$$\mathrm{A} = \begin{bmatrix} 3 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 2 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 2 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 3 \end{bmatrix}.$$

A is the so-called *Wilkinson matrix* and can be generated by the command `wilkinson(7)`. `wilkinson`

Exercise 6.7 By using the Gershgorin circles, provide an estimate of the maximum number of the complex eigenvalues of the following matrices:

$$\mathrm{A} = \begin{bmatrix} 2 & -1/2 & 0 & -1/2 \\ 0 & 4 & 0 & 2 \\ -1/2 & 0 & 6 & 1/2 \\ 0 & 0 & 1 & 9 \end{bmatrix}, \quad \mathrm{B} = \begin{bmatrix} -5 & 0 & 1/2 & 1/2 \\ 1/2 & 2 & 1/2 & 0 \\ 0 & 1 & 0 & 1/2 \\ 0 & 1/4 & 1/2 & 3 \end{bmatrix}.$$

Exercise 6.8 Use the result of Proposition 6.1 to find a suitable shift for the computation of the maximum modulus eigenvalue of

$$\mathrm{A} = \begin{bmatrix} 5 & 0 & 1 & -1 \\ 0 & 2 & 0 & -\frac{1}{2} \\ 0 & 1 & -1 & 1 \\ -1 & -1 & 0 & 0 \end{bmatrix}.$$

Then compare the number of iterations as well the computational cost of the power method both with and without shift by setting the tolerance equal to 10^{-14}.

Exercise 6.9 Show that the matrices $\mathrm{A}^{(k)}$ generated by the QR iteration method are all similar to the matrix A.

Exercise 6.10 Use the command `eig` to compute all the eigenvalues of the two matrices given in Exercise 6.7. Then check how accurate are the conclusions drawn on the basis of Proposition 6.1.

7

Numerical optimization

Let $f : \mathbb{R}^n \to \mathbb{R}$, $n \geq 1$, be a function that we call *cost function* or *objective function*. The problem

$$\min_{\mathbf{x}\in\mathbb{R}^n} f(\mathbf{x}) \tag{7.1}$$

is called unconstrained (or free) optimization problem, whereas

$$\min_{\mathbf{x}\in\Omega} f(\mathbf{x}) \tag{7.2}$$

over a closed proper subset $\Omega \subset \mathbb{R}^n$, is called constrained optimization problem. The set Ω will be determined by either equality or inequality constraints that will be dictated by the nature of the problem to solve. For instance, should we find the optimal allocation of n bounded resources x_i ($i = 1, \ldots, n$), the constraints will be expressed by inequalities as $0 \leq x_i \leq C_i$ (with C_i given constants) and the set Ω will be the subset of $\mathbb{R}^n$ determined by the fulfilment of the constraints

$$\Omega = \{\mathbf{x} = (x_1, \ldots, x_n) :\ 0 \leq x_i \leq C_i,\ i = 1, \ldots, n\}.$$

Since computing the maximum of a function f is equivalent to compute the minimum of $g = -f$, for the sake of simplicity we will only consider algorithms suitable for minimization problems.

Often, more interesting than the minimum value of the given function is the point at which that minimum is achieved, that we call *minimizer*, the latter of course may not be unique.

This chapter will be essentially devoted to numerical methods for unconstrained optimization and, at a lesser extent, to that of constrained optimization.

DOI 10.1007/978-3-642-45367-0_7, © Springer-Verlag Berlin Heidelberg 2014

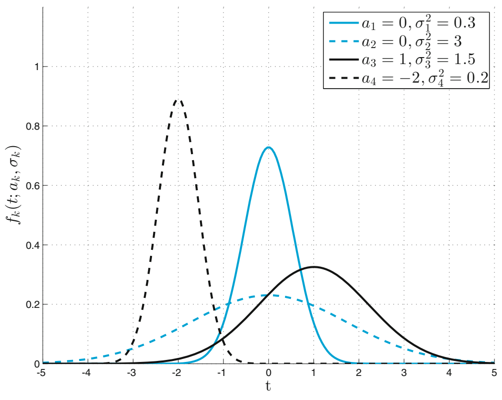


Figure 7.1. Gaussian functions

7.1 Some representative problems

Problem 7.1 (Population dynamics) A colony of 250 bacteries living in an isolated environment self reproduces according to the so called Verhulst model

$$b(t) = \frac{2500}{1 + 9e^{-t/3}}, \quad t > 0$$

where t represents the time (expressed in days) past after the start of the colture ($t = 0$). We would like to find after how many days the population growth rate is maximum. For the solution of this problem, see Examples 7.1 and 7.2. ■

Problem 7.2 (Signal analysis) Applications involving automatic vocal identification, like those implemented on smartphones, compress the acoustic signal into a set of parameters that fully characterize it. The signal intensity is modeled as a sum of m Gaussian functions (also called peaks)

$$f_k(t; a_k, \sigma_k) = \frac{1}{\sqrt{2\pi\sigma_k^2}} e^{-(t-a_k)^2/(2\sigma_k^2)}, \quad \text{for } k = 1, \ldots, m, \ t \in [t_0, t_f], \tag{7.3}$$

characterized by 2 coefficients for every peak: the *expected value* a_k of the kth peak, that is the center of the peak itself, and its *variance* σ_k^2 (see Figure 7.1). The knowledge of the variance of each peak allows the evaluation of both its height $h_k = 1/\sqrt{2\pi\sigma_k^2}$ and amplitude $w_k = 2\sqrt{\log 4\sigma_k^2}$.

We set now

$$f(t; \mathbf{a}, \boldsymbol{\sigma}) = \sum_{k=1}^{m} f_k(t; a_k, \sigma_k), \tag{7.4}$$

where we have set $\mathbf{a} = [a_1, \dots, a_m]$ and $\boldsymbol{\sigma} = [\sigma_1, \dots, \sigma_m]$. The computation of the vectors $\mathbf{a}$ and $\boldsymbol{\sigma}$ entails the solution of the following minimization problem

$$\min_{\mathbf{a},\boldsymbol{\sigma}} \sum_{i=1}^{n} (f(t_i; \mathbf{a}, \boldsymbol{\sigma}) - y_i)^2, \tag{7.5}$$

where (t_i, y_i), $i = 1, \dots, n$ represent a sampling of our signal. (7.5) is a nonlinear least squares problem that is solved in Example 7.12.

The model (7.4) is also named *Gaussian Mixture Model (GMM)* and is used in statistics for *data mining* and *pattern recognition.* ■

Problem 7.3 (Mesh optimization) Consider a given triangulation of the domain $D \subset \mathbb{R}^2$, as in Figure 7.2, left. We want to modify the position of vertices of the internal triangles in order to optimize the triangles' quality in the sense specified below. Let $(x_i^{(k)}, y_i^{(k)})$ (for $i = 0, 1, 2$) be the coordinates of the vertices of the kth triangle T_k. Define the matrix

$$\mathrm{A}_k = \begin{pmatrix} x_1^{(k)} - x_0^{(k)} & x_2^{(k)} - x_0^{(k)} \\ y_1^{(k)} - y_0^{(k)} & y_2^{(k)} - y_0^{(k)} \end{pmatrix}$$

and the scalar quantity

$$\mu_k = \frac{4\det(\mathrm{A}_k)}{\sqrt{3}\|\mathrm{A}_k\mathrm{W}^{-1}\|_F^2}, \tag{7.6}$$

where $\mathrm{W} = [1, 1/2; 0, \sqrt{3}/2]$ while $\|\mathrm{B}\|_F = \sqrt{\sum_{i,j=1}^{2} b_{ij}^2}$ denotes the Frobenius norm of the matrix B. Should T_k be equilateral then $\mu_k = 1$; the more T_k departs from being an equilateral triangle, the more μ_k approaches zero. To optimize our mesh we minimize the function $\sum_{k=1}^{Ne} \mu_k^{-1}$ with respect to the position of the vertices of the internal triangles of D, under the constraints $\det(\mathrm{A}_k) \geq \tau$ (for a given τ), and that no inversion occurs in the ordering of the nodes ([Mun07]). The solution of this problem will be given in Example 7.16.

In Figure 7.2 we display the original triangulation and the optimized one. This kind of problems are faced in the numerical solution of partial differential equations by the finite element method (see Chapter 9). ■

Problem 7.4 (Finance) A given capital is placed in investment funds whose expected rate of interest is 6%, 10%, and 12%, respectively. The risk associated with this investment is modeled by a function that depends on the fractions x_i of the entire capital invested into the three funds, the risk probability of the funds, and their correlations

$$r(\mathbf{x}) = 0.04x_1^2 + 0.25x_2^2 + 0.64x_3^2 + 0.1x_1x_2 + 0.208x_2x_3. \tag{7.7}$$

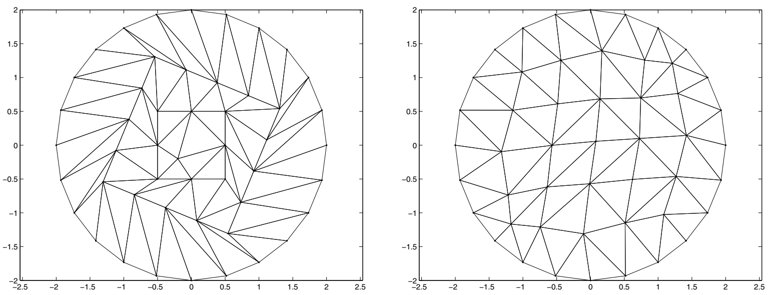


Figure 7.2. Mesh of Problem 7.3. At left the original one, at right that optimized

The goal is to minimize the risk while guaranteeing that the expected return be equal to 10.4%. The solution of this problem is provided in Example 7.14. ■

Problem 7.5 (Road network) Consider a network of n roads and p cross roads as represented in Figure 7.3. Every minute M vehicles travel through the network; on the jth road the maximum speed limit is $v_{j,m}$ km/min, moreover no more than $\rho_{j,m}$ vehicles per km can transit on the jth road s_j.

We aim at finding the optimal density ρ_j (vehicles/km) on the road s_j (with $0 \leq \rho_j \leq \rho_{j,m}$) in order to minimize the average travel time from the inlet (1st node in Figure 7.3) and the outlet (7th node in Figure 7.3). It is assumed that the speed of vehicles traveling along the jth road depends on both the maximum speed $v_{j,m}$ and the maximum density according to the formula $v_j = v_{j,m}(1 - \rho_j/\rho_{j,m})$ km/min. Consequently, the flowrate of vehicles on the jth street is $q_j = \rho_j v_j = \rho_j v_{j,m}(1 - \rho_j/\rho_{j,m})$ vehicles/min. By denoting with t_j (in min) the time necessary to cover the jth road of length L_j km, we find $t_j = L_j/v_j = L_j/(v_{j,m}(1 - \rho_j/\rho_{j,m}))$ min. The objective function to be minimized is

$$f(\boldsymbol{\rho}) = \frac{\sum_{j=1}^{n} t_j \rho_j}{\sum_{j=1}^{n} \rho_j}. \tag{7.8}$$

At every node of the network the vehicles inflow should balance the outflow. By denoting with positive sign those incoming in the ith node ($q_{i,j\text{in}}$) and negative sign those outgoing ($q_{i,j\text{out}}$), the following equations need to be satisfied

$$\sum_{j\text{in}} q_{i,j\text{in}} - \sum_{j\text{out}} q_{i,j\text{out}} = 0 \qquad \text{for } i = 1, \ldots, p. \tag{7.9}$$

This is a constrained minimization problem whose constraints are expressed by both equations and inequalities. See Example 7.17 for its solution. ■

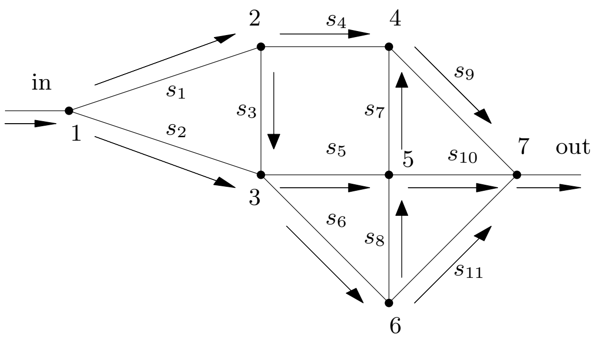


Figure 7.3. The road network of Problem 7.5

7.2 Unconstrained optimization

When minimizing an objective function one might be interested in finding either a local or a global minimizer. A point $\mathbf{x}^* \in \mathbb{R}^n$ is called a *global minimizer* for f if

$$f(\mathbf{x}^*) \leq f(\mathbf{x}) \qquad \forall \mathbf{x} \in \mathbb{R}^n,$$

while $\mathbf{x}^*$ is a *local minimizer* for f if there exists a ball $B_r(\mathbf{x}^*) \subset \mathbb{R}^n$ centered at $\mathbf{x}^*$ and with radius $r > 0$ such that

$$f(\mathbf{x}^*) \leq f(\mathbf{x}) \qquad \forall \mathbf{x} \in B_r(\mathbf{x}^*).$$

We denote by

$$\nabla f(\mathbf{x}) = \left(\frac{\partial f}{\partial x_1}(\mathbf{x}), \ldots, \frac{\partial f}{\partial x_n}(\mathbf{x}) \right)^T \tag{7.10}$$

the *gradient* of f at point $\mathbf{x} \in \mathbb{R}^n$, provided f is differentiable in $\mathbb{R}^n$.

Moreover we denote by $\mathrm{H}(\mathbf{x})$ the *Hessian matrix* of f at the point $\mathbf{x}$, whose elements are

$$h_{ij}(\mathbf{x}) = \frac{\partial^2 f(\mathbf{x})}{\partial x_j \partial x_i}, \qquad i, j = 1, \ldots, n,$$

provided that second derivatives of f at that point $\mathbf{x}$ do exist.

If $f \in C^2(\mathbb{R}^n)$, that is all first and second order derivatives of f exist and are continuous, than $\mathrm{H}(\mathbf{x})$ is symmetric for every $\mathbf{x} \in \mathbb{R}^n$. A point $\mathbf{x}^*$ is called a *stationary (or critical) point* for f if $\nabla f(\mathbf{x}^*) = \mathbf{0}$, a *regular point* if $\nabla f(\mathbf{x}^*) \neq \mathbf{0}$.

A function f defined on $\mathbb{R}^n$ does not necessarily admit a minimizer; moreover, should such point exist, it is not necessarily unique. For instance $f(\mathbf{x}) = x_1 + 3x_2$ is unbounded in $\mathbb{R}^2$, while $f(\mathbf{x}) =$

$\sin(x_1)\sin(x_2)\cdots\sin(x_n)$ admits an infinite number of minimizers and maximizers in $\mathbb{R}^n$ (either local and global).

The function $f : \mathbb{R}^n \to \mathbb{R}$ is *convex* if $\forall \mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ and $\forall \alpha \in [0, 1]$,

$$f(\alpha\mathbf{x} + (1-\alpha)\mathbf{y}) \leq \alpha f(\mathbf{x}) + (1-\alpha)f(\mathbf{y}); \tag{7.11}$$

it is *Lipschitz continuous* if there exists a costant $L > 0$ such that

$$|f(\mathbf{x}) - f(\mathbf{y})| \leq L\|\mathbf{x} - \mathbf{y}\| \qquad \forall \mathbf{x}, \mathbf{y} \in \mathbb{R}^n. \tag{7.12}$$

The following result holds.

Proposition 7.1 (Optimality conditions) *Let $\mathbf{x}^* \in \mathbb{R}^n$. If $\mathbf{x}^*$ is a minimizer for f (either local or global) and if there exists $r > 0$ such that $f \in C^1(B_r(\mathbf{x}^*))$, then $\nabla f(\mathbf{x}^*) = \mathbf{0}$. Moreover, if $f \in C^2(B_r(\mathbf{x}^*))$, the matrix $\mathrm{H}(\mathbf{x}^*)$ is positive semidefinite.*
Viceversa, let $r > 0$ exist such that $f \in C^2(B_r(\mathbf{x}^*))$. If $\nabla f(\mathbf{x}^*) = \mathbf{0}$ and $\mathrm{H}(\mathbf{x})$ is positive definite for all $\mathbf{x} \in B_r(\mathbf{x}^*)$, then $\mathbf{x}^*$ is a local minimizer for f.
Finally, if f is differentiable and convex in $\mathbb{R}^n$ and $\nabla f(\mathbf{x}^*) = \mathbf{0}$, then $\mathbf{x}^*$ is a global minimizer for f.

In view of the numerical solution of optimization problems, an ideal situation is that of a cost function featuring a unique global minimizer. However, most often there exist several local minimizers. This is why in this chapter we will describe numerical methods for the approximation of local minimizers.

Most often, methods for numerical optimization are of iterative type. They may be classified into two categories depending on whether or not they require the knowledge of the derivative of the cost function.

The so called *derivative free* methods make use of direct comparison between values taken by the cost function in order to investigate its local behavior. Among these methods, some make use of numerical approximation (tipically, through finite differences, see Section 9.2.1) of the derivatives, see Section 7.3.

Methods using exact derivatives can take advantage of accurate information on the local function behaviour to achieve faster convergence to the minimizer. As a matter of fact, if f is differentiable at $\overline{\mathbf{x}}$ and $\nabla f(\overline{\mathbf{x}})$ is different than zero, then the maximum growth of f, moving away from $\overline{\mathbf{x}}$, occurs along the (signed) direction given by the vector $\nabla f(\overline{\mathbf{x}})$.

Among the minimization methods that make use of exact derivatives, the two most important classes (based on complementary strategies) are: *descent* (or *line search* methods) and *trust region* methods that will be described in Sections 7.5 and 7.6, respectively.

7.3 Derivative free methods

In this section we describe two simple numerical methods for the minimization of univariate real valued functions or multivariate real valued functions along a one-dimensional direction. We will then describe the Nelder and Mead method for the minimization of functions of several variables.

7.3.1 Golden section and quadratic interpolation methods

Let $f : (a, b) \to \mathbb{R}$ be a continuous function featuring a unique minimizer $x^* \in (a, b)$. We set $I_0 = (a, b)$ and for $k \geq 0$ we generate a sequence of intervals $I_k = (a^{(k)}, b^{(k)})$ of decreasing length, each of those containing x^*.

For any given k, the next interval I_{k+1} is determined as follows. Let $c^{(k)}, d^{(k)} \in I_k$, with $c^{(k)} < d^{(k)}$, be two points such that both

$$\frac{b^{(k)} - a^{(k)}}{d^{(k)} - a^{(k)}} = \frac{d^{(k)} - a^{(k)}}{b^{(k)} - d^{(k)}} = \varphi \tag{7.13}$$

and

$$\frac{b^{(k)} - a^{(k)}}{b^{(k)} - c^{(k)}} = \frac{b^{(k)} - c^{(k)}}{c^{(k)} - a^{(k)}} = \varphi \tag{7.14}$$

be the golden ratio $\varphi = \dfrac{1+\sqrt{5}}{2} \simeq 1.628.$

Thanks to (7.13), (7.14) we find

$$\boxed{c^{(k)} = a^{(k)} + \frac{1}{\varphi^2}(b^{(k)} - a^{(k)}) \quad \text{and} \quad d^{(k)} = a^{(k)} + \frac{1}{\varphi}(b^{(k)} - a^{(k)})} \tag{7.15}$$

The points $c^{(k)}$ and $d^{(k)}$ are symmetrically distributed about the midpoint of I_k, that is

$$\frac{a^{(k)} + b^{(k)}}{2} - c^{(k)} = d^{(k)} - \frac{a^{(k)} + b^{(k)}}{2}. \tag{7.16}$$

Indeed, if we replace $c^{(k)}$ and $d^{(k)}$ in (7.16) with the corresponding expressions given in (7.15), after dividing by the common factor $(b^{(k)} - a^{(k)})/\varphi^2$ we obtain the identity $\varphi^2 - \varphi - 1 = 0$.

Setting $a^{(0)} = a$ and $b^{(0)} = b$, the golden section algorithm is formulated as follows (see Figure 7.4): for $k = 0, 1, \ldots$ until convergence

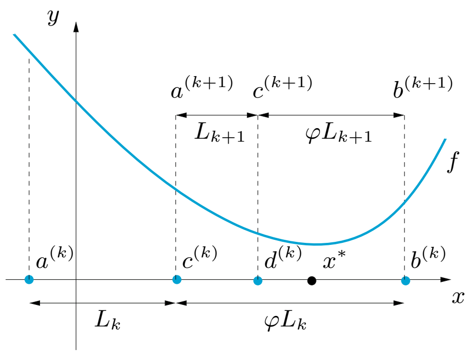


Figure 7.4. A generic iteration of the golden section method for seeking the minimizer of the function f; φ is the golden ratio, while $L_k = c^{(k)} - a^{(k)}$

```
compute $c^{(k)}$ and $d^{(k)}$ through (7.15)
if $f(c^{(k)}) \geq f(d^{(k)})$
   set $I_{k+1} = (a^{(k+1)}, b^{(k+1)}) = (c^{(k)}, b^{(k)})$
else
   set $I_{k+1} = (a^{(k+1)}, b^{(k+1)}) = (a^{(k)}, d^{(k)})$
endif
```
(7.17)

From (7.13) and (7.14) it also follows that $c^{(k+1)} = d^{(k)}$ if $I_{k+1} = (c^{(k)}, b^{(k)})$, while $d^{(k+1)} = c^{(k)}$ if $I_{k+1} = (a^{(k)}, d^{(k)})$.

A stopping criterion for the previous iterations is

$$\boxed{\frac{b^{(k+1)} - a^{(k+1)}}{|c^{(k+1)}| + |d^{(k+1)}|} < \varepsilon} \qquad (7.18)$$

where ε is a given tolerance. In the successful case, the midpoint of the last interval I_{k+1} can be taken as an approximation of the desired minimizer x^*.

Still using (7.13) (or (7.14)) we obtain

$$|b^{(k+1)} - a^{(k+1)}| = \frac{1}{\varphi}|b^{(k)} - a^{(k)}| = \ldots = \frac{1}{\varphi^{k+1}}|b^{(0)} - a^{(0)}|, \qquad (7.19)$$

that is the golden section method converges linearly with a convergence rate $\varphi^{-1} \simeq 0.618$.

This method is implemented in Program 7.1: `fun` is either an *anonymous function* or an *inline function* representing the function f, `a` and `b` are the endpoints of the search interval, `tol` is the tolerance ε and `kmax` is the maximum allowed number of iterations.

The output variable `xmin` contains the value of the minimizer, `fmin` the minimum value of f in (a, b), `iter` the number of iterations carried out by the algorithm.

Program 7.1. golden: golden section method

```
function [xmin,fmin,iter]=golden(fun,a,b,tol,...
                                     kmax,varargin)
%GOLDEN Approximates a minimizer of 1D-functions
%  XMIN=GOLDEN(FUN,A,B,TOL,KMAX) approximates a
%  minimizer of the function FUN in the interval
%  [A,B] by the golden section method.
%  If the search fails, the program returns an
%  error message. FUN is either an anonymous
%  function, or an inline function, or a function
%  defined in a M-file.
%  XMIN=GOLDEN(FUN,A,B,TOL,KMAX,P1,P2,...) passes
%  parameters P1, P2,... to the function
%  FUN(X,P1,P2,...).
%  [XMIN,FMIN,ITER]= GOLDEN(FUN,...) returns
%  the value of FUN at XMIN and the number of
%  iterations required to compute XMIN.
phi=(1+sqrt(5))/2; phi1=1/phi; phi2=1/(phi+1);
c=a+phi2*(b-a); d=a+phi1*(b-a);
err=tol+1; k=0;
while err>tol & k< kmax
  if(fun(c) >= fun(d))
    a=c; c=d; d=a+phi1*(b-a);
  else
    b=d; d=c; c=a+phi2*(b-a);
  end
  k=k+1; err=abs(b-a)/(abs(c)+abs(d));
end
xmin=(a+b)/2; fmin=fun(xmin); iter=k;
if  (iter==kmax & err > tol)
 fprintf(['The golden section method stopped \n',...
 'without converging to the desired tolerance \n',...
 'because the maximum number of iterations was \n',...
 'reached\n']);
end
```

Example 7.1 Let us solve Problem 7.1 using the golden section method. The function $f(t) = -b'(t) = -7500e^{t/3}/(e^{t/3} + 9)^2$ admits a global minimizer in the interval $[6, 7]$ as it appears from its plot. We use Program 7.1 with tolerance equal to 10^{-8} using the following instructions:

```
f=@(t)[-(7500*exp(t/3))/(exp(t/3) + 9)^2]
a=0; b=10; tol=1.e-8; kmax=100;
[tmin,fmin,iter]=golden(f,a,b,tol,kmax)
```

After 38 iterations we find

```
xmin=6.591673759332620        fmin=-2.083333333333333e+02
```

The maximum growth rate is of $208.\overline{3}$ bacteria per day and occurs about after 6.59 days since the start of the colture. ■

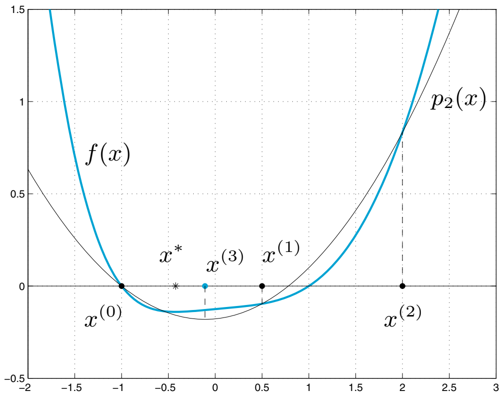


Figure 7.5. The first step of the quadratic interpolation method

The quadratic interpolation method is often used as an alternative to the golden section algorithm.

Let f be a continuous function. Starting from three distinct points $x^{(0)}$, $x^{(1)}$ and $x^{(2)}$, a sequence of points $x^{(k)}$, with $k \geq 3$, is built in a way that $x^{(k+1)}$ represents the vertex (and therefore the minimizer) of the parabola $p_2^{(k)}$ interpolating f at the points $x^{(k)}$, $x^{(k-1)}$, and $x^{(k-2)}$, see Figure 7.5:

$$p_2^{(k)}(x) = f(x^{(k-2)}) + f[x^{(k-2)}, x^{(k-1)}](x - x^{(k-2)}) + \\ f[x^{(k-2)}, x^{(k-1)}, x^{(k)}](x - x^{(k-2)})(x - x^{(k-1)}).$$

Here,

$$f[x_i, x_j] = \frac{f(x_j) - f(x_i)}{x_j - x_i}, \quad f[x_i, x_j, x_\ell] = \frac{f[x_j, x_\ell] - f[x_i, x_j]}{x_\ell - x_i} \tag{7.20}$$

are the so called *Newton divided differences* (see [QSS07, Ch. 8]). By solving the first order equation ${p_2^{(k)}}'(x^{(k+1)}) = 0$ we obtain

$$\boxed{x^{(k+1)} = \frac{1}{2}\left(x^{(k-2)} + x^{(k-1)} - \frac{f[x^{(k-2)}, x^{(k-1)}]}{f[x^{(k-2)}, x^{(k-1)}, x^{(k)}]}\right)} \tag{7.21}$$

We iterate until $|x^{(k+1)} - x^{(k)}| < \varepsilon$ for a prescribed tolerance $\varepsilon > 0$.

Provided for every k the divided difference $f[x^{(k-2)}, x^{(k-1)}, x^{(k)}]$ does not vanish, this method converges super-linearly to the minimizer with a convergence rate $p \simeq 1.3247$ (see [Bre02]). Otherwise, the method may not terminate. For this reason the quadratic interpolation method is tipically used in combination with other methods, such as the golden section method, whose convergence is always guaranteed.

The command MATLAB `fminbnd` implements the combination of these two methods. The calling sintax is `x = fminbnd(fun,a,b)` where `fun` is the function handle associated with the cost function and `a, b` represent the endpoints of the interval containing the minimizer. The output `x` provides the approximation of the minimizer.

`fminbnd`

Example 7.2 We use function `fminbnd` to solve the same problem described in Example 7.1. We use the following commands:

```
a=0; b=10; tol=1.e-8; kmax=100;
[tmin1,fmin1,exitflag,output]=fminbnd(f,a,b,...
                    optimset('TolX',1.e-8));
```

Convergence to `fmin1= 6.591673708945312` is achieved in 8 iterations, much fewer than the 38 iterations requested by the golden section method. The command `optimset` allows fixing the tolerance to a desired value (`tol=1.e-8` in the current case) different than the one that would be otherwise set by default (`tol=1.e-4`). The output optional parameters are: `fmin1` containing the evaluation of f at the minimizer, `exitflag` indicating the termination state, and `output` containing the number of iterations carried out as well as the global number of function evaluations requested by the whole algorithm. ■

`optimset`

As noticed, the two previous methods are genuinely one dimensional, yet they can be used to solve multidimensional optimization problems provided they are restricted to the search of optimizers along a given one dimensional direction (see Section 7.5).

7.3.2 Nelder and Mead method

Let $n > 1$ and $f : \mathbb{R}^n \to \mathbb{R}$ be a continuous function.

The $n-$*simplex* with $n+1$ vertices $\mathbf{x}_i \in \mathbb{R}^n$ (with $i = 0, \dots, n$) is the set

$$S = \{\mathbf{y} \in \mathbb{R}^n : \ \mathbf{y} = \sum_{i=0}^{n} \lambda_i \mathbf{x}_i, \ \lambda_i \in \mathbb{R} \text{ and } \lambda_i \geq 0 : \ \sum_{i=0}^{n} \lambda_i = 1\}, \quad (7.22)$$

under the assumption that the n vectors $\mathbf{x}_i - \mathbf{x}_0$ ($i = 1, \dots, n$) be linearly independent (S is a segment in $\mathbb{R}$, a triangle in $\mathbb{R}^2$ and a tethraedron in $\mathbb{R}^3$).

The method of Nelder and Mead [NM65] is a derivative free minimization algorithm which generates a sequence of simplices $\{S^{(k)}\}_{k\geq 0}$ in $\mathbb{R}^n$ that run after or circumscribe the minimizer $\mathbf{x}^* \in \mathbb{R}^n$ of the cost function f. It uses the evaluations of f at the simplices' vertices, as well as simple geometrical transformations such as reflections, expansions and contractions.

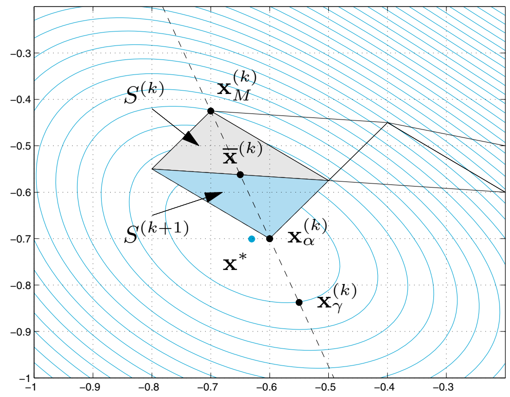


Figure 7.6. One step of the Nelder and Mead method, the point $\mathbf{x}_M^{(k)}$ is replaced by $\mathbf{x}_\alpha^{(k)}$

To generate the initial simplex $S^{(0)}$ we take a point $\tilde{\mathbf{x}} \in \mathbb{R}^n$ and a positive real number η, and set $\mathbf{x}_i^{(0)} = \tilde{\mathbf{x}} + \eta \mathbf{e}_i$ for $i = 0, \ldots, n$, where $\{\mathbf{e}_i\}$ are the vectors of the standard basis in $\mathbb{R}^n$.

For every $k \geq 0$ (until convergence) we select the "worst" vertex of $S^{(k)}$

$$\mathbf{x}_M^{(k)} = \underset{0 \leq i \leq n}{\operatorname{argmax}}\, f(\mathbf{x}_i^{(k)}) \tag{7.23}$$

then replace it by a new point to form the new simplex $S^{(k+1)}$.

The new point is chosen as follows. First we select

$$\mathbf{x}_m^{(k)} = \underset{0 \leq i \leq n}{\operatorname{argmin}}\, f(\mathbf{x}_i^{(k)}) \quad \text{and} \quad \mathbf{x}_\mu^{(k)} = \underset{\substack{0 \leq i \leq n \\ i \neq M}}{\operatorname{argmax}}\, f(\mathbf{x}_i^{(k)}) \tag{7.24}$$

and define the *centroid* of the hyperplane $H^{(k)}$ passing through the vertices $\{\mathbf{x}_i^{(k)},\ i = 0, \ldots, n,\ i \neq M\}$

$$\overline{\mathbf{x}}^{(k)} = \frac{1}{n} \sum_{\substack{i=0 \\ i \neq M}}^{n} \mathbf{x}_i^{(k)}. \tag{7.25}$$

(When $n = 2$, the centroid is the midpoint of the edge of $S^{(k)}$ opposite to $\mathbf{x}_M^{(k)}$, see Fig. 7.6.)

Then we compute the reflection $\mathbf{x}_\alpha^{(k)}$ of $\mathbf{x}_M^{(k)}$ with respect to the hyperplane $H^{(k)}$, i.e.

$$\mathbf{x}_\alpha^{(k)} = (1 - \alpha)\overline{\mathbf{x}}^{(k)} + \alpha \mathbf{x}_M^{(k)}, \tag{7.26}$$

where $\alpha < 0$ is the reflection coefficient (tipically, $\alpha = -1$). The point $\mathbf{x}_\alpha^{(k)}$ lies on the straight line joining $\overline{\mathbf{x}}^{(k)}$ and $\mathbf{x}_M^{(k)}$, on the side of $\overline{\mathbf{x}}^{(k)}$ far from $\mathbf{x}_M^{(k)}$ (see Fig. 7.6).

Now, we compare $f(\mathbf{x}_\alpha^{(k)})$ with the values of f at the other vertices of the simplex, before accepting $\mathbf{x}_\alpha^{(k)}$ as the new vertex. Meanwhile, we try to move $\mathbf{x}_\alpha^{(k)}$ on the straight line joining $\overline{\mathbf{x}}^{(k)}$ and $\mathbf{x}_M^{(k)}$, to set the new simplex $S^{(k+1)}$. More precisely we proceed as follows.

- If $f(\mathbf{x}_\alpha^{(k)}) < f(\mathbf{x}_m^{(k)})$, i.e. the reflection has produced a new minimum, we compute

$$\mathbf{x}_\gamma^{(k)} = (1-\gamma)\overline{\mathbf{x}}^{(k)} + \gamma \mathbf{x}_M^{(k)}, \tag{7.27}$$

  where $\gamma < -1$ (tipically, $\gamma = -2$). Then, if $f(\mathbf{x}_\gamma^{(k)}) < f(\mathbf{x}_m^{(k)})$, $\mathbf{x}_M^{(k)}$ is replaced by $\mathbf{x}_\gamma^{(k)}$; otherwise $\mathbf{x}_M^{(k)}$ is replaced by $\mathbf{x}_\alpha^{(k)}$; then we proceed by incrementing k by one.
- If $f(\mathbf{x}_m^{(k)}) \le f(\mathbf{x}_\alpha^{(k)}) < f(\mathbf{x}_\mu^{(k)})$, $\mathbf{x}_M^{(k)}$ is replaced by $\mathbf{x}_\alpha^{(k)}$; then we proceed by incrementing k by one.
- If $f(\mathbf{x}_\mu^{(k)}) \le f(\mathbf{x}_\alpha^{(k)}) < f(\mathbf{x}_M^{(k)})$ we compute

$$\mathbf{x}_\beta^{(k)} = (1-\beta)\overline{\mathbf{x}}^{(k)} + \beta \mathbf{x}_\alpha^{(k)}, \tag{7.28}$$

  where $\beta > 0$ (tipically, $\beta = 1/2$). Now, if $f(\mathbf{x}_\beta^{(k)}) > f(\mathbf{x}_M^{(k)})$ define the vertices of the new simplex $S^{(k+1)}$ by

$$\mathbf{x}_i^{(k+1)} = \frac{1}{2}(\mathbf{x}_i^{(k)} + \mathbf{x}_m^{(k)}) \tag{7.29}$$

  otherwise $\mathbf{x}_M^{(k)}$ is replaced by $\mathbf{x}_\beta^{(k)}$; then we proceed by incrementing k by one.
- If $f(\mathbf{x}_\alpha^{(k)}) > f(\mathbf{x}_M^{(k)})$ we compute

$$\mathbf{x}_\beta^{(k)} = (1-\beta)\overline{\mathbf{x}}^{(k)} + \beta \mathbf{x}_M^{(k)}, \tag{7.30}$$

  (again $\beta > 0$), if $f(\mathbf{x}_\beta^{(k)}) > f(\mathbf{x}_M^{(k)})$ define the vertices of the new simplex $S^{(k+1)}$ by (7.29), otherwise $\mathbf{x}_M^{(k)}$ is replaced by $\mathbf{x}_\beta^{(k)}$; then we proceed by incrementing k by one.

As soon as the stopping criterium $\max\limits_{i=0,\dots,n} \|\mathbf{x}_i^{(k)} - \mathbf{x}_m^{(k)}\|_\infty < \varepsilon$ is fulfilled, $\mathbf{x}_m^{(k)}$ will be retained as an approximation of the minimizer.

The convergence of Nelder and Mead method is guaranteed in very special cases only (see example [LRWW99]); in fact a stagnation may occur in which case the algorithm needs to be restarted. Nevertheless, this algorithm is quite robust and efficient for small dimensional problems. Its rate of convergence is severely affected by the choice of the initial simplex. The Nelder and Mead method is implemented by the MATLAB command `fminsearch`; its sintax is described in the following example. `fminsearch`

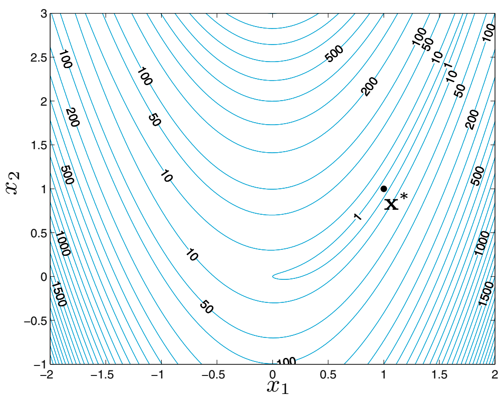


Figure 7.7. Contour lines of the Rosenbrock function

Example 7.3 (The Rosenbrock function) The Rosenbrock function

$$f(x_1, x_2) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2,$$

whose contour lines are displayed in Figure 7.7 ([Ros61]), is often used to test both efficiency and robustness of minimization algorithms. Its global minimum is attained at the point $\mathbf{x}^* = (1, 1)$, however its variation around $\mathbf{x}^*$ is fairly low, making algorithms' convergence quite problematic. Through the following command

```
fun=@(x) 100*(x(2)-x(1)^2)^2+(1-x(1))^2; x0=[-1.2,1]
xstar=fminsearch(fun,x0)
```

we get

```
xstar =
    1.000022021783570      1.000042219751772
```

In MATLAB, by replacing the second instruction with the expanded one

```
[xstar,fval,exitflag,output]=fminsearch(fun,x0)
```

we would obtain additional information on the minimum value of f `fval=8.1777e-10`, on the number of iterations, `output.iterations=85` as well as the total number of function evaluations `output.funcCount=159`. Finally, the error tolerance can be modified by using the command `optimset`, as already discussed in Example 7.2. ■

See Exercises 7.1-7.3.

7.4 The Newton method

Assume that $f : \mathbb{R}^n \to \mathbb{R}$ $(n \geq 1)$ is of class $C^2(\mathbb{R}^n)$ and that we know how to compute its first and second order partial derivatives. We can apply Newton's method, already introduced in Chapter 2 for the solution of the system $\mathbf{F}(\mathbf{x}) = \nabla f(\mathbf{x}) = \mathbf{0}$, whose Jacobian matrix $J_{\mathbf{F}}(\mathbf{x}^{(k)})$ is nothing but the Hessian matrix of $\mathbf{F}$ computed at the generic iteration point $\mathbf{x}^{(k)}$. The method reads as follows: for a given $\mathbf{x}^{(0)} \in \mathbb{R}^n$, for $k = 0, 1, \ldots$, until convergence

$$\boxed{\begin{array}{ll} \text{solve} & \mathrm{H}(\mathbf{x}^{(k)})\boldsymbol{\delta}\mathbf{x}^{(k)} = -\nabla f(\mathbf{x}^{(k)}) \\ \text{set} & \mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \boldsymbol{\delta}\mathbf{x}^{(k)} \end{array}} \tag{7.31}$$

For a given tolerance $\varepsilon > 0$, a suitable stopping test is $\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\| \leq \varepsilon$.

Example 7.4 The function

$$f(\mathbf{x}) = \frac{2}{5} - \frac{1}{10}(5x_1^2 + 5x_2^2 + 3x_1x_2 - x_1 - 2x_2)e^{-(x_1^2+x_2^2)} \tag{7.32}$$

is displayed in Figure 7.8, right. We apply Newton's method to approximate its global minimizer $\mathbf{x}^* \simeq (-0.63065832, -0.7007420)$ (rounded to the first 7 significant digits). We take $\mathbf{x}^{(0)} = (-0.9, -0.9)$ and tolerance $\varepsilon = 10^{-5}$. After 5 iterations the method (7.31) converges to `x=[-0.63058;-0.70074]`. Should we have chosen $\mathbf{x}^{(0)} = (-1, -1)$, after 400 iterations the stopping test would not be fulfilled. In fact a necessary condition for convergence of Newton's method is that $\mathbf{x}^{(0)}$ should be sufficiently close to the minimizer $\mathbf{x}^*$ (see Section 2.3). This is known as *local convergence* of Newton's method.

The reader should be aware that Newton's method may converge to any stationary point (not necessarily to a minimizer). For instance, by taking $\mathbf{x}^{(0)} = (0.5, -0.5)$ after 5 iterations the method converges to the saddle point `x=[0.80659; -0.54010]`. ■

A general convergence criterium for the method (7.31) is as follows: if $f \in C^2(\mathbb{R}^n)$, $\mathbf{x}^*$ is a stationary point, the Hessian matrix $\mathrm{H}(\mathbf{x}^*)$ is positive definite, the components of $\mathrm{H}(\mathbf{x})$ are Lipschitz continuous in a neighbourhood of $\mathbf{x}^*$ and $\mathbf{x}^{(0)}$ is sufficiently close to $\mathbf{x}^*$, then the Newton method (7.31) converges quadratically to $\mathbf{x}^*$ (see, for instance [SY06, pag. 132], [NW06]).

In spite of its simple implementation, Newton's method is computationally demanding when n is large (as it requires the analytic expression of the derivatives and, at each iteration, the computation of both the gradient and the Hessian matrix of f). Besides, $\mathbf{x}^{(0)}$ has to be chosen close enough to $\mathbf{x}^*$.

A suitable strategy to build up efficient and robust minimization algorithms relies on combining locally convergent with globally convergent methods, as described in the next section.

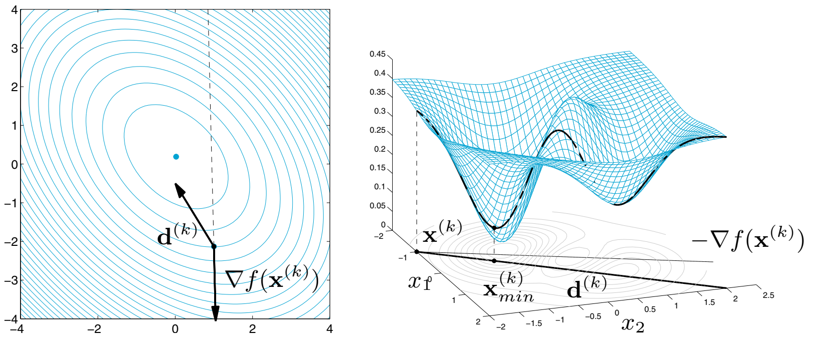


Figure 7.8. At left, countour lines of a function $f(\mathbf{x})$, its gradient evaluated at $\mathbf{x}^{(k)}$ and a suitable descent direction $\mathbf{d}^{(k)}$. At right, the restriction of the function $f(\mathbf{x})$ (7.32) along a descent direction $\mathbf{d}^{(k)}$ and the minimizer $\mathbf{x}^{(k)}_{min}$ along $\mathbf{d}^{(k)}$

7.5 Descent (or line search) methods

In this Section we assume for simplicity that $f \in C^2(\mathbb{R})$ and is bounded from below.

Descent methods (also known as line search methods) are iterative methods in which, for every $k \geq 0$, $\mathbf{x}^{(k+1)}$ depends on $\mathbf{x}^{(k)}$, on a vector $\mathbf{d}^{(k)}$ depending on $\nabla f(\mathbf{x}^{(k)})$ and on a suitable parameter $\alpha_k \in \mathbb{R}$. Given $\mathbf{x}^{(0)} \in \mathbb{R}^n$, the method reads as follows:
for $k = 0, 1, \ldots,$ until convergence

$$\boxed{\begin{array}{l}\text{find a direction } \mathbf{d}^{(k)} \in \mathbb{R}^n\\ \text{compute the step } \alpha_k \in \mathbb{R}\\ \text{set } \mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{d}^{(k)}\end{array}} \tag{7.33}$$

The vector $\mathbf{d}^{(k)}$ must be a *descent direction*, meaning that

$$\begin{array}{ll}\mathbf{d}^{(k)^T}\nabla f(\mathbf{x}^{(k)}) < 0 & \text{if } \nabla f(\mathbf{x}^{(k)}) \neq \mathbf{0},\\ \mathbf{d}^{(k)} = \mathbf{0} & \text{if } \nabla f(\mathbf{x}^{(k)}) = \mathbf{0}.\end{array} \tag{7.34}$$

The name *descent direction* arises from the property that the vector $\nabla f(\mathbf{x}^{(k)})$ provides in $\mathbb{R}^n$ the direction with sign of maximum positive growth of f moving from $\mathbf{x}^{(k)}$. As $\mathbf{d}^{(k)^T}\nabla f(\mathbf{x}^{(k)})$ represents the directional derivative of f along $\mathbf{d}^{(k)}$, the first condition in (7.34) ensures that we are moving along a direction opposite to the gradient, that is towards a minimizer of f, as displayed in Figure 7.8.

Some popular descent directions will be reported in the next Section.

Once $\mathbf{d}^{(k)}$ is determined, the optimum value of $\alpha_k \in \mathbb{R}$ is the one that guarantees the maximum variation of f along $\mathbf{d}^{(k)}$ and can therefore be computed by solving a one-dimensional minimization problem (that is minimizing the restriction of f along $\mathbf{d}^{(k)}$), see Figure 7.8.

However, as the computation of α_k is quite involved when f is not a quadratic function, we will report in Section 7.5.2 some alternative techniques aimed at approximating α_k.

7.5.1 Descent directions

The most widely used descent directions are:

1. *Newton's directions*

$$\boxed{\mathbf{d}^{(k)} = -(\mathrm{H}(\mathbf{x}^{(k)}))^{-1}\nabla f(\mathbf{x}^{(k)})} \tag{7.35}$$

2. *quasi-Newton directions*

$$\boxed{\mathbf{d}^{(k)} = -\mathrm{H}_k^{-1}\nabla f(\mathbf{x}^{(k)})} \tag{7.36}$$

   where H_k represents an approximation of the true Hessian matrix $\mathrm{H}(\mathbf{x}^{(k)})$. This choice is a valuable alternative to Newtons' method when second derivatives of f are heavy to compute (see Section 7.5.4);
3. *gradient directions*

$$\boxed{\mathbf{d}^{(k)} = -\nabla f(\mathbf{x}^{(k)})} \tag{7.37}$$

   (they can be regarded as a trivial example of quasi-Newton directions);
4. *conjugate gradient directions*

$$\boxed{\begin{aligned}&\mathbf{d}^{(0)} = -\nabla f(\mathbf{x}^{(0)})\\&\mathbf{d}^{(k+1)} = -\nabla f(\mathbf{x}^{(k+1)}) - \beta_k \mathbf{d}^{(k)}, \ k \geq 0\end{aligned}} \tag{7.38}$$

   The coefficients β_k can be chosen according to different criteria, see Section 7.5.5, however they coincide with those of Conjugate Gradient method for linear systems (see (5.66)) when f is a quadratic function.

The descent direction (7.37) fulfills the condition (7.34), then (7.35) and (7.36) assure that $\mathrm{H}(\mathbf{x}^{(k)})$ and H_k, respectively, are positive definite matrices. The vectors (7.38) fulfill (7.34) provided that the coefficients β_k are suitably chosen, as we will see in Section 7.5.5.

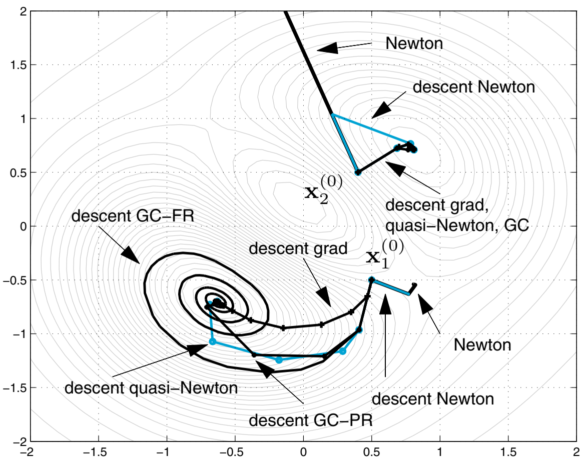


Figure 7.9. Convergence history of Newton's and descent methods for the function of the Example 7.5

Example 7.5 Consider again the function $f(\mathbf{x})$ (7.32), featuring two local minimizers, one local maximizer and two saddle points. See Figure 7.8, right. We compare the sequences $\{\mathbf{x}^{(k)}\}$ generated by Newton's method (7.31) and descent methods with descent directions given by (7.35)–(7.38).

Consider first $\mathbf{x}_1^{(0)} = (0.5, -0.5)$ as initial point. In Figure 7.9 we see that Newton's method (7.31) converges to the saddle point $(.8065, -.5401)$; the descent method with Newton direction (7.35) breaks down at the second iteration as it generates a matrix $\mathrm{H}(\mathbf{x}^{(1)})$ which is not definite positive. (See Remark 7.2 on how to overcome this drawback.) The other descent methods with directions given by (7.36), (7.37), and (7.38) (for the latter, two different criteria for the determination of the parameters β_k have been used, named GC-FR and GC-PR, see Section 7.5.5) converge to the local minimizer $(-0.6306, -0.7007)$. The faster convergence is achieved in 9 iterations using quasi-Newton directions (7.36), see the blue path in Figure 7.9. By choosing a different initial point $\mathbf{x}_2^{(0)} = (0.4, 0.5)$, the Newton method diverges while method (7.35), even though it shares the same first descent direction with Newton's method, builds up a short steplength α_k which then allows convergence to the local minimizer $(0.8095, 0.7097)$ in only 4 iterations. All the other descent methods with directions (7.36), (7.37), and (7.38) converge in 10 to 15 iterations to the same local minimizer. ■

The choice of the steplength α_k will be discussed in Section 7.5.2, while the analysis of different descent directions is deferred to Sections 7.5.3–7.5.5.

7.5.2 Strategies for choosing the steplength α_k

Once the descent direction $\mathbf{d}^{(k)}$ is chosen, the steplength α_k has to be determined in such a way that the new iterate $\mathbf{x}^{(k+1)}$ be the minimizer of f along such a direction, that is

$$\alpha_k = \underset{\alpha \in \mathbb{R}}{\operatorname{argmin}}\; f(\mathbf{x}^{(k)} + \alpha \mathbf{d}^{(k)}).$$

A second order Taylor expansion of f around $\mathbf{x}^{(k)}$ yields

$$\begin{aligned} f(\mathbf{x}^{(k)} + \alpha \mathbf{d}^{(k)}) = f(\mathbf{x}^{(k)}) + \alpha {\mathbf{d}^{(k)}}^T \nabla f(\mathbf{x}^{(k)}) + \\ \frac{\alpha^2}{2} {\mathbf{d}^{(k)}}^T \mathrm{H}(\mathbf{x}^{(k)}) \mathbf{d}^{(k)} + o(\|\alpha \mathbf{d}^{(k)}\|^2). \end{aligned} \tag{7.39}$$

In the special case in which f is a quadratic function, that is

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T \mathrm{A}\mathbf{x} - \mathbf{x}^T \mathbf{b} + c$$

with $\mathrm{A} \in \mathbb{R}^{n\times n}$, $\mathbf{b} \in \mathbb{R}^n$ symmetric and positive definite and $c \in \mathbb{R}$, the expansion in (7.39) is exact, that is the infinitesimal residual is null. As $\mathrm{H}(\mathbf{x}^{(k)}) = \mathrm{A}$ for every $k \geq 0$ and $\nabla f(\mathbf{x}^{(k)}) = \mathrm{A}\mathbf{x}^{(k)} - \mathbf{b} = -\mathbf{r}^{(k)}$ (see (5.35)), by differentiating (7.39) with respect to α and setting the derivative equal to zero we obtain

$$\boxed{\alpha_k = \frac{{\mathbf{d}^{(k)}}^T \mathbf{r}^{(k)}}{{\mathbf{d}^{(k)}}^T \mathrm{A}\mathbf{d}^{(k)}}} \tag{7.40}$$

In the case (7.37), we find $\mathbf{d}^{(k)} = \mathbf{r}^{(k)}$ thus we obtain the well known gradient method described in Chapter 5, which obeys the convergence estimate (5.59).

Instead, should the direction $\mathbf{d}^{(k)}$ be chosen as in (7.38), by setting

$$\beta_k = -\frac{(\mathrm{A}\mathbf{d}^{(k)})^T \mathbf{r}^{(k+1)}}{{\mathbf{d}^{(k)}}^T \mathrm{A}\mathbf{d}^{(k)}} \tag{7.41}$$

we would recover the conjugate gradient method (5.66) for linear systems which fulfills the convergence estimate (5.67).

If f is a generic (non quadratic) function, the computation of the optimal α_k would require an iterative method to solve numerically the above minimization problem along the direction $\mathbf{d}^{(k)}$. In these cases an approximate (rather than exact) value of α_k can be chosen by requiring that the new iterate $\mathbf{x}^{(k+1)}$ defined in (7.33) ensures that

$$f(\mathbf{x}^{(k+1)}) < f(\mathbf{x}^{(k)}). \tag{7.42}$$

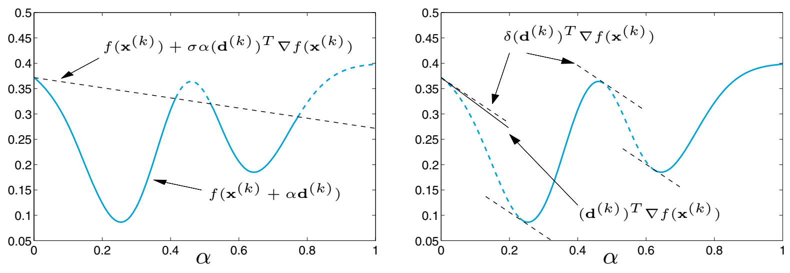


Figure 7.10. At left, the terms comparing in the first inequality in (7.43) when $\sigma = 0.2$. $(7.43)_1$ is satisfied for those values of α providing the continuous lightblue line. At right, some straightlines with slope $\delta\mathbf{d}^{(k)^T}\nabla f(\mathbf{x}^{(k)})$ and $\delta = 0.9$, $(7.43)_2$ is fulfilled for those α corresponding to the continuous lightblue line. The Wolfe conditions are simultaneously fulfilled when either $0.23 \leq \alpha \leq 0.41$ or $0.62 \leq \alpha \leq 0.77$

In this respect, a natural strategy is that of assigning α_k a large value and then reducing it iteratively until when (7.42) is satisfied. Unfortunately, this strategy does not guarantee that the associated sequence $\{\mathbf{x}^{(k)}\}$ converges to the desired minimizer $\mathbf{x}^*$. See Exercise 7.4 and the associated Figure 10.8, left, where steplengths are too long. See also Exercise 7.5 and the associated Figure 10.8, right, where steplengths are now too short.

A better criterium for the choice of $\alpha_k > 0$ is the one based on the *Wolfe's conditions*:

$$\boxed{\begin{array}{l} f(\mathbf{x}^{(k)} + \alpha_k\mathbf{d}^{(k)}) \leq f(\mathbf{x}^{(k)}) + \sigma\alpha_k\mathbf{d}^{(k)^T}\nabla f(\mathbf{x}^{(k)}) \\ \mathbf{d}^{(k)^T}\nabla f(\mathbf{x}^{(k)} + \alpha_k\mathbf{d}^{(k)}) \geq \delta\mathbf{d}^{(k)^T}\nabla f(\mathbf{x}^{(k)}) \end{array}} \tag{7.43}$$

where σ and δ, such that $0 < \sigma < \delta < 1$, are two given constants and $\mathbf{d}^{(k)^T}\nabla f(\mathbf{x}^{(k)})$ represents the directional derivative of f along the direction $\mathbf{d}^{(k)}$.

The first inequality in (7.43) is named *Armijo's rule*, and it inhibits too little variations of f with respect to the steplength α_k (see Figure 7.10, left). More precisely, the larger α_k the higher the variation of f.

The second Wolfe condition states that at the new point $\mathbf{x}^{(k)}+\alpha_k\mathbf{d}^{(k)}$ the value of the directional derivative of f should be larger than δ times the same derivative at the previous value $\mathbf{x}^{(k)}$ (see Figure 7.10, right).

From the example depicted in Figure 7.10 one can see that Wolfe's conditions might also be fulfilled far from the minimizer of f along $\mathbf{d}^{(k)}$ and even when the directional derivative of f takes large values. More restricitive conditions than (7.43) are the *strong Wolfe's conditions*

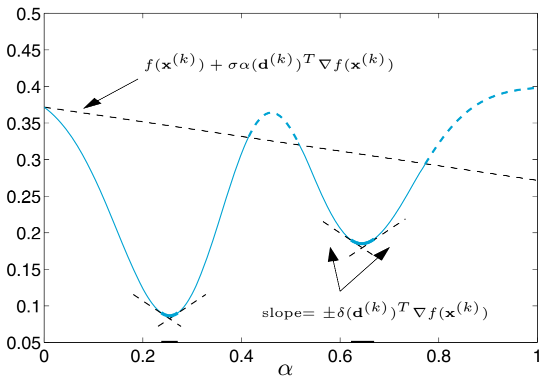


Figure 7.11. The strong Wolfe's conditions (7.44) are fulfilled when α belongs to small intervals around the minimizers, in correspondence with the thick lightblue piece of curve. $\sigma = 0.2$ and $\delta = 0.9$ have been considered.

$$\boxed{\begin{aligned} & f(\mathbf{x}^{(k)} + \alpha_k\mathbf{d}^{(k)}) \le f(\mathbf{x}^{(k)}) + \sigma\alpha_k\mathbf{d}^{(k)T}\nabla f(\mathbf{x}^{(k)}), \\ & |\mathbf{d}^{(k)T}\nabla f(\mathbf{x}^{(k)} + \alpha_k\mathbf{d}^{(k)})| \le -\delta\mathbf{d}^{(k)T}\nabla f(\mathbf{x}^{(k)}) \end{aligned}} \qquad (7.44)$$

being $0 < \sigma < \delta < 1$ suitable fixed constants.

The first condition is the same as in (7.43), whereas the second one gives rise to $(7.43)_2$ as well as to $\mathbf{d}^{(k)T}\nabla f(\mathbf{x}^{(k)} + \alpha_k\mathbf{d}^{(k)}) \le -\delta\mathbf{d}^{(k)T}\nabla f(\mathbf{x}^{(k)})$ (having recalled that the right hand side of $(7.44)_2$ is positive because of $(7.34)_1$). Conditions $(7.44)_2$ inhibits f to vary too strongly at $\mathbf{x}^{(k)} + \alpha_k\mathbf{d}^{(k)}$ (see Figure 7.11 for an example).

It can be proved (see, e.g., [NW06, Lemma 3.1]) that if $\mathbf{d}^{(k)}$ is a descent direction in $\mathbf{x}^{(k)}$ and $f \in C^1(\mathbb{R}^n)$ is lower bounded in the set $\{\mathbf{x}^{(k)} + \alpha\mathbf{d}^{(k)},\ \alpha > 0\}$, then for every σ, δ such that $0 < \sigma < \delta < 1$, there exist intervals of steplengths α_k satisfying (7.43) and (7.44).

In practice, σ is chosen very small, e.g. $\sigma = 10^{-4}$ ([NW06]), while δ is large ($\delta = 0.9$) for Newton, quasi-Newton and gradient directions, small ($\delta = 0.1$) for the conjugate gradient directions.

A simple strategy to determine the steplength α_k satisfying Wolfe's conditions is *backtracking*: it consists of starting with $\alpha = 1$ and then reducing it by a prescribed factor ρ (tipically, $\rho \in [1/10, 1/2)$) until when the first condition (7.43) is satisfied. In pseudocode: for a given $\mathbf{x}^{(k)}$ and a descent direction $\mathbf{d}^{(k)}$, for $\sigma \in (0, 1)$, $\rho \in [1/10, 1/2)$

```
set $\alpha = 1$
while $f(\mathbf{x}^{(k)} + \alpha\mathbf{d}^{(k)}) > f(\mathbf{x}^{(k)}) + \sigma\alpha{\mathbf{d}^{(k)}}^T\nabla f(\mathbf{x}^{(k)})$
    $\alpha = \alpha\rho$
end
set $\alpha_k = \alpha$
```
(7.45)

The second condition in (7.43) is never checked because the backtracking technique intrinsically computes steplengths that are not too small.

Remark 7.1 The backtracking technique is often combined with replacing f by a quadratic or cubic interpolant of f along $\mathbf{d}^{(k)}$. The chosen steplength α_k yields a new point $\mathbf{x}^{(k+1)}$ which represents the minimizer of the interpolant of f along $\mathbf{d}^{(k)}$. The corresponding algorithm is named quadratic or cubic line search, respectively. See [NW06, Ch. 3] for further details on this approach. ■

The Program `backtrack` 7.2 implements the strategy (7.45). Parameters `fun` and `grad` are function handles rispectively associated with the functions $f(\mathbf{x})$ and $\nabla f(\mathbf{x})$; `xk` and `dk` respectively contain the point $\mathbf{x}^{(k)}$ and the descent direction $\mathbf{d}^{(k)}$, while `sigma` and `rho` contain the parameter values σ and ρ. When `sigma` and `rho` are not specified, the default values $\sigma = 10^{-4}$ and $\rho = 1/4$ are set. The output variable `x` contains the new point $\mathbf{x}^{(k+1)}$.

Program 7.2. backtrack: backtracking strategy

```
function [x,alphak]= backtrack(fun,xk,gk,dk,varargin)
%BACKTRACK Backtracking strategy for line search.
%   [X,ALPHAK] = BACKTRACK(FUN,XK,GK,DK) computes the
%   new point x_{k+1}=x_k+alpha_k d_k, where alpha_k
%   is determined by the backtracking technique
%   with sigma=1.e-4 and rho=1/4.
%   [X,ALPHAK] = BACKTRACK(FUN,XK,GK,DK,SIGMA,RHO)
%   allows to specify the parameters sigma and rho.
%   Tipically 1.e-4<sigma<0.1 and 1/10< rho <1/2.
%   FUN is the function handle associated with the cost
%   function. XK, GK, and DK contain respectively the
%   point x_k, the gradient of f at x_k and the
%   descent direction d_k.
if nargin==4
    sigma=1.e-4; rho=1/4;
else
    sigma=varargin{1}; rho=varargin{2};
end
alphamin=1.e-5; % minimum value allowed for alpha_k
alphak = 1; fk = fun(xk);
k=0; x=xk+alphak*dk;
while fun(x)>fk+sigma*alphak*gk'*dk & alphak>alphamin
    alphak = alphak*rho;
    x = xk+alphak*dk; k = k+1;
end
```

The Program descent 7.3 implements the descent method (7.33) with directions (7.35)–(7.38) and steplengths α_k determined according to the backtracking strategy. The stopping criterium is (see [JS96])

$$\boxed{\max_{1\le i\le n}\left|\frac{[\nabla f(\mathbf{x}^{(k+1)})]_i \max\{|\mathbf{x}_i^{(k+1)}|,\mathrm{typ}(\mathbf{x}_i)\}}{\max\{|f(\mathbf{x}^{(k+1)})|,\mathrm{typ}(f(\mathbf{x}))\}}\right| \le \varepsilon} \qquad (7.46)$$

for a given $\varepsilon > 0$, where typ(x) is a characteristic value expressing the order of magnitude of the x variable. Its presence prevents test failure when either $\mathbf{x}^*$ or $f(\mathbf{x}^*)$ are null.

Parameters fun and grad are function handles associated with $f(\mathbf{x})$ and $\nabla f(\mathbf{x})$, respectively, x0 contains the initial value of the sequence, tol the tolerance of the stopping criterium and kmax the maximum allowed number of iterations. The variable meth sorts the descent direction: Newton's directions correspond to meth=1, quasi-Newton's to meth=2, gradient directions to meth=3, while meth=41, 42, 43 select three different directions of the conjugate gradient: CG-FR, CG-PR, and CG-HS, respectively, as we will see in Section 7.5.5.

Program 7.3. descent: descent method

```
function [x,err,iter]= descent(fun,grad,x0,tol,kmax,...
                              meth,varargin)
%DESCENT Descent method for optimization
%  [X,ERR,ITER]=DESCENT(FUN,GRAD,X0,TOL,KMAX,METH,HESS)
%  computes a local minimizer of function FUN by the
%  descent method with Newton directions (METH=1),
%  quasi-Newton directions (BFGS) (METH=2), gradient
%  directions (METH=3) or conjugate gradient directions
%  with Fletcher and Reeves beta_k (METH=41),
%  Polak and Ribiere beta_k (METH=42),
%  Hestenes and Stiefel beta_k (METH=43).
%  The steplength is computed by the backtracking
%  technique.  FUN, GRAD and HESS (the latter being
%  used only if METH=1) are function handles associated
%  with the cost function, its gradient and its Hessian
%  matrix, respectively. If METH=2, HESS is a matrix
%  approximating the Hessian of FUN at the initial
%  point X0. TOL is the tolerance for the stopping
%  test, while KMAX is the maximum allowed number of
%  iterations. The function backtrack is called inside.
if nargin>6
if meth==1, hess=varargin{1};
elseif meth==2, H=varargin{1}; end
end
err=tol+1; k=0; xk=x0(:); gk=grad(xk); dk=-gk;
eps2=sqrt(eps);
while err>tol & k< kmax
if meth==1;          H=hess(xk); dk=-H\gk; % Newton
elseif meth==2       dk=-H\gk;             % BFGS
elseif meth==3       dk=-gk;               % gradient
```

```
end
[xk1,alphak]= backtrack(fun,xk,gk,dk);
gk1=grad(xk1);
if meth==2 % BFGS update
  yk=gk1-gk; sk=xk1-xk; yks=yk'*sk;
  if yks> eps2*norm(sk)*norm(yk)
  Hs=H*sk;
  H=H+(yk*yk')/yks-(Hs*Hs')/(sk'*Hs);
  end
elseif meth>=40 % CG update
  if meth == 41
   betak=-(gk1'*gk1)/(gk'*gk); % FR
  elseif meth == 42
   betak=-(gk1'*(gk1-gk))/(gk'*gk); % PR
  elseif meth == 43
   betak=-(gk1'*(gk1-gk))/(dk'*(gk1-gk)); % HS
  end
  dk=-gk1-betak*dk;
end
xk=xk1; gk=gk1; k=k+1; xkt=xk1;
for i=1:length(xk1); xkt(i)=max([abs(xk1(i)),1]); end
err=norm((gk1.*xkt)/max([abs(fun(xk1)),1]),inf);
end
x=xk; iter=k;
if (k==kmax & err > tol)
 fprintf(['Descent method stopped \n',...
 'without converging to the desired tolerance \n',...
 'because the maximum number of iterations was \n',...
 'reached\n']);
end
```

Example 7.6 Consider again function $f(\mathbf{x})$ (7.32). To approximate its global minimizer $(-0.6306, -0.7007)$, we use the `diff` command introduced in Section 1.5.3 for the symbolic computation of both the gradient of f and the Hessian matrix H of f. Then we define the function handles `f`, `grad_f`, and `hess` respectively associated with f, ∇f, and H and call the Program 7.3 with the following instructions:

```
x0=[0.5;-0.5]; tol=1.e-5; kmax=200;
meth=1;  % Newton's directions
[x1,err1,k1]= descent(f,grad_f,x0,tol,kmax,meth,hess);
meth=2; hess=eye(2); % quasi-Newton directions
[x2,err2,k2]= descent(f,grad_f,x0,tol,kmax,meth,hess);
meth=3; % gradient directions
[x3,err3,k3]= descent(f,grad_f,x0,tol,kmax,meth);
meth=42; % CG-PR directions
[x4,err4,k4]= descent(f,grad_f,x0,tol,kmax,meth);
```

We choose $\mathbf{x}^{(0)} = (0.5, -0.5)$, tolerance 10^{-5} and maximum number of iterations equal to 200 and obtain these results:

```
descent Newton k=200,      x=[ 7.7015e-01,-6.3212e-01]
descent quasi-Newton k=9,  x=[-6.3058e-01,-7.0075e-01]
descent gradient k=17,     x=[-6.3058e-01,-7.0075e-01]
descent CG-PR k=17,        x=[-6.3060e-01,-7.0073e-01]
```

Note that the descent method with Newton's direction has not achieved convergence because directions can be generated that do not fulfill condition (7.34). ■

In the next sections we indicate how to compute the approximate Hessian matrices H_k and the parameters β_k in (7.36) and (7.38). Moreover, we will comment on the convergence properties of the various methods introduced so far.

7.5.3 The descent method with Newton's directions

Consider a lower bounded function $f \in C^2(\mathbb{R}^n)$ and the descent method (7.33) with Newton's descent directions (7.35) and steplengths α_k fulfilling the Wolfe's conditions (7.43).

Assume that for every $k \geq 0$ the Hessian matrix $\mathrm{H}(\mathbf{x}^{(k)})$ in (7.35), besides being symmetric thanks to the assumption on f, is positive definite. Moreover, setting $\mathrm{B}_k = \mathrm{H}(\mathbf{x}^{(k)})$ we suppose that

$$\exists M > 0 : \; K(\mathrm{B}_k) = \|\mathrm{B}_k\| \|\mathrm{B}_k^{-1}\| \leq M \qquad \forall k \geq 0. \tag{7.47}$$

(Note that $K(\mathrm{B}_k)$ coincides with the spectral condition number of B_k, see (5.31).)

Then the sequence $\mathbf{x}^{(k)}$ generated by (7.33) converges to a stationary point $\mathbf{x}^*$ of f. Moreover, by choosing $\alpha_k = 1$ from a given $\overline{k}$ on (that is when we are sufficiently close to $\mathbf{x}^*$) the convergence order is quadratic. See [NW06, Thm. 3.2] for the proof.

Remark 7.2 Since the Hessian matrices are positive definite, the stationary point $\mathbf{x}^*$ must necessarily be a minimizer.

However, should $\mathrm{H}(\mathbf{x}^{(k)})$ fail to be positive definite for a given k, the corresponding $\mathbf{d}^{(k)}$ in (7.35) could fail to be a descent direction and the Wolfe conditions might become meaningless. To overcome this problem the Hessian matrix could be replaced by $\mathrm{B}_k = \mathrm{H}(\mathbf{x}^{(k)}) + \mathrm{E}_k$ for a suitable matrix E_k (either diagonal or not) in such a way that B_k is positive definite and $\mathbf{d}^{(k)} = -\mathrm{B}_k^{-1}\nabla f(\mathbf{x}^{(k)})$ turns out to be a descent direction. ■

The descent method with Newton's directions is implemented in Program 7.3.

Example 7.7 Let us compute the global minimizer of the function $f(\mathbf{x})$ (7.32) by using the descent method (7.33), with the Newton's directions (7.35) and steplengths α_k satisfying the Wolfe conditions. We use a tolerance $\varepsilon = 10^{-5}$ for the stopping criterium and we start from $\mathbf{x}^{(0)} = (-1, -1)$. By using Program 7.3 with `meth=1`, after 4 iterations, we have convergence to `x=[-0.63058;-0.70074]`. Choosing instead $\mathbf{x}^{(0)} = (0.5, -0.5)$, the method stagnates as $\mathrm{H}(\mathbf{x}^{(0)})$ is not positive definite, yielding a vector $\mathbf{d}^{(0)}$ which is not a descent direction; consequently, the backtracking technique is unable to find a value $\alpha_0 > 0$ that fulfills the Wolfe conditions. ■

7.5.4 Descent methods with quasi-Newton directions

When using the directions (7.36) we need a strategy to build H_k. For a given symmetric and positive definite matrix H_0, a popular recursive technique is that based on the so called *rank-one update* of Broyden's method (2.19) for the solution of nonlinear systems. The matrices H_k are required:

– to satisfy the secant condition

$$\mathrm{H}_{k+1}(\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}) = \nabla f(\mathbf{x}^{(k+1)}) - \nabla f(\mathbf{x}^{(k)});$$

– to be symmetric, as $\mathrm{H}(\mathbf{x})$;
– to be positive definite to guarantee that vectors $\mathbf{d}^{(k)}$ are descent directions;
– to satisfy the condition

$$\lim_{k\to\infty} \frac{\|(\mathrm{H}_k - \mathrm{H}(\mathbf{x}^*))\mathbf{d}^{(k)}\|}{\|\mathbf{d}^{(k)}\|} = 0,$$

which, from one hand, ensures that H_k is a good approximation of $\mathrm{H}(\mathbf{x}^*)$ along the descent direction $\mathbf{d}^{(k)}$ and, from the other hand, guarantees a super-linear rate of convergence.

The strategy due to Broyden, Fletcher, Goldfarb, and Shanno (BFGS) is based on the following recursivity relationship

$$\boxed{\mathrm{H}_{k+1} = \mathrm{H}_k + \frac{\mathbf{y}^{(k)}\mathbf{y}^{(k)T}}{\mathbf{y}^{(k)T}\mathbf{s}^{(k)}} - \frac{\mathrm{H}_k\mathbf{s}^{(k)}\mathbf{s}^{(k)T}\mathrm{H}_k}{\mathbf{s}^{(k)T}\mathrm{H}_k\mathbf{s}^{(k)}}} \tag{7.48}$$

where $\mathbf{s}^{(k)} = \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}$ and $\mathbf{y}^{(k)} = \nabla f(\mathbf{x}^{(k+1)}) - \nabla f(\mathbf{x}^{(k)})$. These matrices are symmetric and positive definite under the condition $\mathbf{y}^{(k)T}\mathbf{s}^{(k)} > 0$, which is fulfilled provided the steplengths α_k satisfy the Wolfe conditions (either (7.43) or (7.44)). See [JS96].

The corresponding BFGS method (implemented in Program 7.3) can be summarized as follows: for a given $\mathbf{x}^{(0)} \in \mathbb{R}^n$ and a suitable symmetric and positive definite matrix $\mathrm{H}_0 \in \mathbb{R}^{n\times n}$ which approximates $\mathrm{H}(\mathbf{x}^{(0)})$, for $k = 0, 1, \ldots,$ until convergence:

$$\boxed{\begin{array}{ll}
\text{solve} & \mathrm{H}_k\mathbf{d}^{(k)} = -\nabla \mathbf{f}(\mathbf{x}^{(k)}) \\
\text{compute} & \alpha_k \text{ satisfying Wolfe's conditions} \\
\text{set} & \mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k\mathbf{d}^{(k)} \\
 & \mathbf{s}^{(k)} = \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)} \\
 & \mathbf{y}^{(k)} = \nabla f(\mathbf{x}^{(k+1)}) - \nabla f(\mathbf{x}^{(k)}) \\
\text{compute} & \mathrm{H}_{k+1} \text{ using (7.48)}
\end{array}} \tag{7.49}$$

Under the condition that $f \in C^2(\mathbb{R}^n)$ is lower bounded and the matrices H_k are positive definite with a condition number uniformly bounded (see (7.47)), the BFGS method converges to a minimizer with (superlinear) convergence order $p \in (1, 2)$ (see for instance [JS96, NW06]).

Example 7.8 We apply the BFGS method (7.49) to compute the minimizer of the (yet another time) function $f(\mathbf{x})$ (7.32). We choose $\varepsilon = 10^{-5}$ for the stopping criterium and H_0 equal to the identity matrix (which is obviously symmetric and positive definite). The latter choice is more convenient than choosing $\mathrm{H}_0 = \mathrm{H}(\mathbf{x}^{(0)})$, i.e. the exact Hessian in $\mathbf{x}^{(0)}$, as it yields a faster convergence. Program 7.3 with `meth=2` and `hess=eye(2)` converges to `x=[-0.63058;-0.70074]` in 6 iterations if $\mathbf{x}^{(0)} = (-1, -1)$ and in 9 iterations if $\mathbf{x}^{(0)} = (0.5, -0.5)$. ■

Remark 7.3 As in Broyden method (2.19), the computational cost of order $\mathcal{O}(n^3)$ for the calculation of $\mathbf{d}^{(k)} = -\mathrm{H}_k^{-1}\nabla f(\mathbf{x}^{(k)})$ can be reduced to order $\mathcal{O}(n^2)$, by using QR factorizations of H_k (see [GM72]).
An alternative strategy is based on the use of the inverse $\widetilde{\mathrm{H}}_k$ of H_k both in (7.48) and (7.49). This strategy can be implemented in order of $\mathcal{O}(n^2)$ operations per step, however in practice it is less stable than the more standard (7.48). ■

The BFGS method (as well as several other minimization methods) is implemented in the MATLAB function `fminunc` included in the *optimization toolbox*. By the following commands:

fminunc

```
fun=@(x) 100*(x(2)-x(1)^2)^2+(1-x(1))^2; x0=[1.2;-1];
options = optimset('LargeScale','off');
[x,fval,exitflag,output]=fminunc(fun,x0,options)
```

the function `fminunc` computes the minimizer of the Rosenbrock function using the BFGS method (which corresponds to using the value `'off'` to initialize the option `'LargeScale'`). The output parameters have the same meaning as those of the function `fminsearch` described in Example 7.3. Convergence is achieved in 24 iterations with a tolerance $\varepsilon = 10^{-6}$; this has required 93 function evaluations.

With the previous options the gradient of the function f is approximated in `fminunc` by using finite difference methods (see Section 9.2.1). However, in case an exact expression of the gradient of f is available, it can be passed to the function as follows:

```
fun=@(x) 100*(x(2)-x(1)^2)^2+(1-x(1))^2; x0=[1.2;-1];
grad_fun=@(x)[-400*(x(2)-x(1)^2)*x(1)-2*(1-x(1));
   200*(x(2)-x(1)^2)];
options = optimset('LargeScale','off','GradObj','on');
[x,fval,exitflag,output]=fminunc({fun,grad_fun},...
                         x0,options)
```

Note the changement in the command `options`. Convergence is achieved in 25 iterations with 32 function evaluations.

bfgsmin

Octave 7.1 The BFGS method is implemented in the Octave function `bfgsmin`. The Octave command `fminunc` instead implements the *trust region* method that we describe in Section 7.6. ■

7.5.5 Gradient and conjugate gradient descent methods

Let us consider the descent method (7.33) with gradient directions (7.37). As already noticed, the latter are descent directions.

If $f \in C^2(\mathbb{R}^n)$ is lower bounded and the steplengths α_k satisfy Wolfe's conditions, this method converges linearly to a steady point ([NW06]). See Program 7.3 for its implementation.

Example 7.9 We consider once more the function (7.32). We fix the tolerance $\varepsilon = 10^{-5}$ for the stopping criterium and call Program 7.3 setting `meth=3` (this corresponds to gradient directions). Choosing $\mathbf{x}^{(0)} = (-0.9, -0.9)$, $\mathbf{x}^{(0)} = (-1, -1)$ and $\mathbf{x}^{(0)} = (0.5, -0.5)$, the method converges to the global minimizer `x=[-0.63058;-0.70074]` in 11, 12, and 17 iterations, respectively. Choosing instead $\mathbf{x}^{(0)} = (0.9, 0.9)$, which is closer to the local minimizer $\mathbf{x}^* = (.8094399, .7097390)$, the method converges to the latter in 21 iterations. ■

Consider now the conjugate gradient directions (7.38). Several options are available for the choice of β_k (see for instance [SY06, NW06]). Among those we quote the following:

1. *Fletcher–Reeves (1964)*

$$\boxed{\beta_k^{FR} = -\frac{\|\nabla f(\mathbf{x}^{(k)})\|^2}{\|\nabla f(\mathbf{x}^{(k-1)})\|^2}} \tag{7.50}$$

2. *Polak–Ribière (1969) (also known as Polak–Ribière–Polyak parameters)*

$$\boxed{\beta_k^{PR} = -\frac{\nabla f(\mathbf{x}^{(k)})^T(\nabla f(\mathbf{x}^{(k)}) - \nabla f(\mathbf{x}^{(k-1)}))}{\|\nabla f(\mathbf{x}^{(k-1)})\|^2}} \tag{7.51}$$

3. *Hestenes–Stiefel (1952)*

$$\boxed{\beta_k^{HS} = -\frac{\nabla f(\mathbf{x}^{(k)})^T(\nabla f(\mathbf{x}^{(k)}) - \nabla f(\mathbf{x}^{(k-1)}))}{\mathbf{d}^{(k-1)^T}(\nabla f(\mathbf{x}^{(k)}) - \nabla f(\mathbf{x}^{(k-1)}))}} \tag{7.52}$$

In fact, all these choices reduce to (7.41) if f is a quadratic convex function.

For coherence, we will indicate with FR (respectively, PR, HS) the directions associated with β_k^{FR} (respectively, β_k^{PR}, β_k^{HS}).

The following are sufficient conditions for the FR conjugate gradient converge to a steady point ([NW06, SY06]): $f \in C^1(\mathbb{R}^n)$, its gradient is Lipschitz continuous, the initial point $\mathbf{x}^{(0)}$ is such that the set $A = \{\mathbf{x} : f(\mathbf{x}) \leq f(\mathbf{x}^{(0)})\}$ is bounded and the steplengths α_k satisfy the strong Wolfe's conditions (7.44) with $0 < \sigma < \delta < 1/2$.

Under the same assumptions on f and $\mathbf{x}^{(0)}$ and under the condition that β_k^{PR} is replaced by $\beta_k^{PR+} = \max\{-\beta_k^{PR}, 0\}$ also the PR conjugate gradient method with these modified coefficients converges to a steady point, provided however that the steplengths α_k undergo a variant of the strong Wolfe's conditions (7.44). Same conclusions hold for the HS conjugate gradient algorithm. We refer to [Noc92, NW06, SY06] for the proof and a more in-depth analysis.

The conjugate gradient method with FR, PR, and HS directions and steplengths α_k computed by the *backtracking* technique are all implemented in Program 7.3.

Example 7.10 Still on the function (7.32) we fix a tolerance $\varepsilon = 10^{-5}$ for the stopping criterium and call Program 7.3 by setting `meth=41, 42, 43`, which correspond to the conjugate gradient method associated with directions FR, PR, and HS, respectively. The number of iterations are reported in the table below.

Directions		$\mathbf{x}^{(0)}$	
	$(-1,-1)$	$(1,1)$	$(0.5,-0.5)$
FR	20	12	>400
PR	21	28	17
HS	23	40	28

For both choices $\mathbf{x}^{(0)} = (-1,-1)$ and $\mathbf{x}^{(0)} = (0.5,-0.5)$, the method converges to the global minimizer `x=[-0.63058;-0.70074]`, whereas with $\mathbf{x}^{(0)} = (1,1)$ all the variants converge to the local minimizer `x=[0.8094;0.7097]`. ■

Several remarks are in order.

From the previous table and Fig. 7.9, we see that directions PR and HS are more efficient than FR. The latter may be quite inefficient and generate very tiny steplengths. This may yield very slow convergence or even stagnation; in the latter case the algorithm can be restarted by using a gradient direction $\mathbf{d}^{(k)} = -\nabla f(\mathbf{x}^{(k)})$.

When the steplengths α_k are computed exactly (as described at the beginning of Sect. 7.5.1) the rate of convergence of the conjugate gradient method is simply linear, that of Newton methods quadratic, while that of quasi-Newton's super-linear. In spite of that, the conjugate gradient method is simple to implement: it does not require the Hessian matrix (neither its approximations) and only one evaluation of f and its gradient is required at every iteration. It is definitely preferable on large dimensional optimization problems, whereas Newton and quasi-Newton methods are in general more efficient on small dimensional problems.

See Exercises 7.4-7.6.

7.6 Trust region methods

At the generic kth step, line search methods determine the descent direction $\mathbf{d}^{(k)}$ first and then the steplength α_k. Instead trust region methods choose direction and steplength simultaneously by building a ball centered at $\mathbf{x}^{(k)}$ and radius δ_k (the so called trust region), a quadratic approximation $\tilde{f}_k$ of the objective function f and choosing the new value $\mathbf{x}^{(k+1)}$ as the minimizer of $\tilde{f}_k$ in the trust region, see Figure 7.12.

More precisely, we start by a "trust" value $\delta_k > 0$, we use second order Taylor development of f about $\mathbf{x}^{(k)}$ to compute $\tilde{f}_k$,

$$\tilde{f}_k(\mathbf{s}) = f(\mathbf{x}^{(k)}) + \mathbf{s}^T\nabla f(\mathbf{x}^{(k)}) + \frac{1}{2}\mathbf{s}^T\mathrm{H}_k\mathbf{s} \qquad \forall\mathbf{s}\in\mathbb{R}^n \tag{7.53}$$

where H_k is either Hessian of f at $\mathbf{x}^{(k)}$ or a suitable approximation of it, then we compute

$$\mathbf{s}^{(k)} = \underset{\mathbf{s}\in\mathbb{R}^n:\ \|\mathbf{s}\|\le\delta_k}{\mathrm{argmin}} \quad \tilde{f}_k(\mathbf{s}). \tag{7.54}$$

At this stage we compute

$$\rho_k = \frac{f(\mathbf{x}^{(k)}+\mathbf{s}^{(k)}) - f(\mathbf{x}^{(k)})}{\tilde{f}_k(\mathbf{s}^{(k)}) - \tilde{f}_k(\mathbf{0})}, \tag{7.55}$$

then we proceed as follows:
i) If ρ_k is close to one, we accept $\mathbf{s}^{(k)}$ and move to the next iteration. However, if the minimizer of $\tilde{f}_k$ lies on the border of the trust region, we extend the latter before proceeding with the next iteration.
ii) If ρ_k is either negative or positive and small (much smaller than one), we reduce the trust region and look for a new $\mathbf{s}^{(k)}$ by solving again problem (7.54).
iii) Finally if ρ_k is much larger than one, we accept $\mathbf{s}^{(k)}$, we keep the trust region as is, and move to the next iteration.

Should the second derivative of f be available we could take H_k equal to the Hessian (or, in case the latter fails to be positive definite, one of its variants described in Remark 7.2). Otherwise, H_k can be built recursively as done for quasi-Newton descent direction method (see Sect. 7.5.4).

Assume that: H_k is symmetric positive definite and $\|\mathrm{H}_k^{-1}\nabla f(\mathbf{x}^{(k)})\| \le \delta_k$; then (7.54) admits $\mathbf{s}^{(k)} = \mathrm{H}_k^{-1}\nabla f(\mathbf{x}^{(k)})$ as minimizer in the trust region. Otherwise the minimizer of $\tilde{f}_k$ lies at the exterior of the trust region; in that case one has to solve a minimization problem for $\tilde{f}_k$ constrained to the circumference centered at $\mathbf{x}^{(k)}$ with radius δ_k, that is

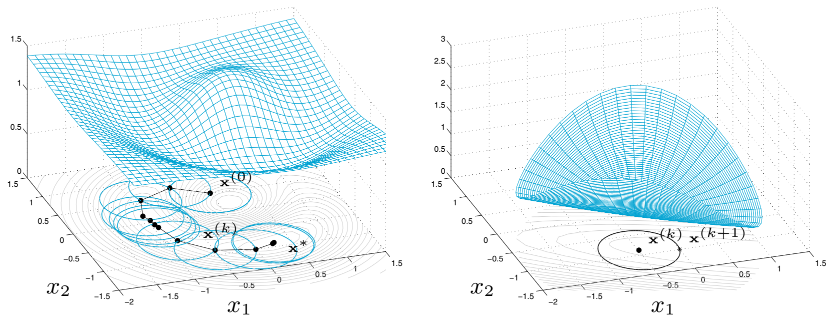


Figure 7.12. Convergence history of trust region method *(at left)* and the quadratic model $\tilde{f}_k$ at step $k = 8$ *(at right)*

$$\min_{\mathbf{s}\in\mathbb{R}^n:\ \|\mathbf{s}\|=\delta_k} \tilde{f}_k(\mathbf{s}). \tag{7.56}$$

To solve (7.56) we can use the Lagrange multipliers approach (see Section 7.8.2), that is we look for the saddle point of the Lagrangian $\mathcal{L}_k(\mathbf{s},\lambda) = \tilde{f}_k(\mathbf{s})+\frac{1}{2}\lambda(\mathbf{s}^T\mathbf{s}-\delta_k^2)$, i.e. for a vector $\mathbf{s}^{(k)}$ and a scalar $\lambda^{(k)} > 0$ satisfying:

$$\begin{array}{l}(\mathrm{H}_k + \lambda^{(k)}\mathrm{I})\mathbf{s}^{(k)} = -\nabla f(\mathbf{x}^{(k)}),\\ (\mathrm{H}_k + \lambda^{(k)}\mathrm{I}) \text{ is semidefinite positive}\\ \|\mathbf{s}^{(k)}\| - \delta_k = 0.\end{array} \tag{7.57}$$

From $(7.57)_1$ we formally derive $\mathbf{s}^{(k)} = \mathbf{s}^{(k)}(\lambda^{(k)})$ and we replace it into $(7.57)_3$ to get the nonlinear equation

$$\varphi(\lambda^{(k)}) = \frac{1}{\|\mathbf{s}^{(k)}(\lambda^{(k)})\|} - \frac{1}{\delta_k} = 0.$$

The reason for using instead of $(7.57)_3$ its reciprocal is that the latter is easier to solve numerically. Indeed few Newton iterations (tipically, 3 or less) suffice. Precisely, for a given $\lambda_0^{(k)}$, setting $\mathbf{g}^{(k)} = \nabla f(\mathbf{x}^{(k)})$, we proceed as follows:

```
for $\ell = 0, \ldots, 2$
    compute $\mathbf{s}_\ell^{(k)} = -(\mathrm{H}_k + \lambda_\ell^{(k)}\mathrm{I})^{-1}\mathbf{g}^{(k)}$
    evaluate $\varphi(\lambda_\ell^{(k)}) = \frac{1}{\|\mathbf{s}_\ell^{(k)}\|} - \frac{1}{\delta_k}$
    evaluate $\varphi'(\lambda_\ell^{(k)})$
    update $\lambda_{\ell+1}^{(k)} = \lambda_\ell^{(k)} - \frac{\varphi(\lambda_\ell^{(k)})}{\varphi'(\lambda_\ell^{(k)})}$
```

The vector $\mathbf{s}_\ell^{(k)}$ is obtained by using the Cholesky factorization (5.18) of $\mathrm{B}_\ell^{(k)} = (\mathrm{H}_k + \lambda_\ell^{(k)}\mathrm{I})$ provided this matrix is positive definite. (Notice that $\mathrm{B}_\ell^{(k)}$ is symmetric, in view of the definition of H_k, and its eigenvalues are all real.) More in general, instead of $\mathrm{B}_\ell^{(k)}$ we use $(\mathrm{B}_\ell^{(k)} + \beta\mathrm{I})$ where β is larger than the negative eigenvalue of maximum modulus of $\mathrm{B}_\ell^{(k)}$.

By suitably representing the derivative of $\varphi(\lambda^{(k)})$, problem (7.54) can be solved by using the following algorithm: for $\mathbf{g}^{(k)} = \nabla f(\mathbf{x}^{(k)})$ and a given δ_k,

```
solve $\mathrm{H}_k\mathbf{s} = -\mathbf{g}^{(k)}$
if $\|\mathbf{s}\| \leq \delta_k$ and $\mathrm{H}_k$ is positive definite
   set $\mathbf{s}^{(k)} = \mathbf{s}$
else
   compute $\beta_1 =$ the negative eigenvalue of $\mathrm{H}_k$
      with largest modulus
   set $\lambda_0^{(k)} = 2|\beta_1|$
   for $\ell = 0, \ldots, 2$                                          (7.58)
      compute R :  $\mathrm{R}^T\mathrm{R} = \mathrm{H}_k + \lambda_\ell^{(k)}\mathrm{I}$
      solve $\mathrm{R}^T\mathrm{R}\mathbf{s} = -\mathbf{g}^{(k)},\ \mathrm{R}^T\mathbf{q} = \mathbf{s}$
      update   $\lambda_{\ell+1}^{(k)} = \lambda_\ell^{(k)} + \left(\frac{\|\mathbf{s}\|}{\|\mathbf{q}\|}\right)^2 \frac{\|\mathbf{s}\| - \delta_k}{\delta_k}$
   set $\mathbf{s}^{(k)} = \mathbf{s}$
endif
```

In conclusion, we provide the trust region algorithm in its simplest form for the solution of the minimization problem (7.1) ([CL96a, CL96b]). Consider an initial point $\mathbf{x}^{(0)}$, a maximum value $\hat{\delta} > 0$ for the trust region radii and an initial radius $0 < \delta_0 < \hat{\delta}$. Consider then four real parameters η_1, η_2, γ_1 and γ_2 such that $0 < \eta_1 < \eta_2 < 1$ and $0 < \gamma_1 < 1 < \gamma_2$ for updating the trust region and a real parameter $0 \leq \mu < \eta_1$ for the acceptability of the solution. For $k = 0, 1, \ldots,$ until convergence

```
compute $f(\mathbf{x}^{(k)})$, $\nabla f(\mathbf{x}^{(k)})$ and $\mathrm{H}_k$,
solve $\min_{\|\mathbf{s}\|_2 \leq \delta_k} \tilde{f}_k(\mathbf{s})$ by (7.58)
compute $\rho_k$ using (7.55),
if $\rho_k > \mu$
    set $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \mathbf{s}^{(k)}$
else
    set $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)}$
endif
if $\rho_k < \eta_1$
    set $\delta_{k+1} = \gamma_1 \delta_k$
elseif $\eta_1 \leq \rho_k \leq \eta_2$
    set $\delta_{k+1} = \delta_k$
elseif $\rho_k > \eta_2$ and $\|\mathbf{s}^{(k)}\| = \delta_k$
    set $\delta_{k+1} = \min\{\gamma_2 \delta_k, \hat{\delta}\}$
endif
```
(7.59)

A possible choice of parameters is $\eta_1 = 1/4$, $\eta_2 = 3/4$, $\gamma_1 = 1/4$, $\gamma_2 = 2$ (see [NW06]). By choosing $\mu = 0$ we accept any step yielding a decrease of f; choosing instead $\mu > 0$ we only accept steps for which the variation of f be at least μ times that of its quadratic model $\tilde{f}_k$.

Remark 7.4 (Approximate solution of (7.54)) Problem (7.54) can be solved approximately, using however an approximation that does not affect the convergence properties of the trust region method. A possible strategy consists in solving the problem not in the whole $\mathbb{R}^n$ but rather in a subspace of dimension two. More precisely, we look for the solution of

$$\min_{\mathbf{s} \in \mathcal{S}_k:\ \|\mathbf{s}\| \leq \delta_k} \tilde{f}_k(\mathbf{s}). \tag{7.60}$$

If H_k is positive (or negative) definite, $\mathcal{S}_k = \mathrm{span}\{\nabla f(\mathbf{x}^{(k)}), \mathrm{H}_k^{-1} \nabla f(\mathbf{x}^{(k)})\}$; otherwise we compute its negative eigenvalue β_1 with maximum modulus and set $\mathcal{S}_k = \mathrm{span}\{\nabla f(\mathbf{x}^{(k)}), (\mathrm{H}_k + \alpha I)^{-1} \nabla f(\mathbf{x}^{(k)})\}$, with $\alpha \in (-\beta_1, -2\beta_1]$. The choice of these subspaces is motivated by the search of the so-called Cauchy point, the minimizer of $\tilde{f}_k$ along the directional gradient and internal to the trust region ([NW06]). The most demanding computational effort when solving (7.60) consists in the factorization of either H_k or $\mathrm{H}_k + \alpha I$ and in computing its eigenvalue β_1. However, the computational cost required by (7.60) is definitely lower than that necessary to solve (7.54). ■

The algorithm (7.59) is implemented in Program 7.4. Parameters `fun, grad, x0, tol, kmax` have the same meaning as in the Program `descent` 7.3. Moreover, `delta0` is the radius of the initial trust region, meth characterizes the choice of matrices H_k: if meth=1, hess contains the function handle of the Hessian of f and H_k is the exact Hessian. If meth is different than one there is no need to pass the input variable hess; in this case H_k is a rank-one approximation of the Hessian computed as in (7.48).

Program 7.4. trustregion: trust region method

```
function [x,err,iter]= trustregion(fun,grad,x0,...
                  delta0,tol,kmax,meth,hess)
%TRUSTREGION Trust region method for minimization
%  [X,ERR,ITER]=TRUSTREGION(FUN,GRAD,X0,TOL,KMAX,...
%  METH,HESS) computes a local minimizer of function
%  f by the trust region method. FUN and GRAD
%  (and HESS) are the function handles of the cost
%  function, its gradient (and its Hessian).
%  If METH=1, the Hessian HESS of f is used, otherwise
%  rank-one updates approximations of the Hessian are
%  built as in BFGS and the variable HESS is not requi-
%  red. X0 is the initial point for the sequence gene-
%  rated by the method. TOL is the tolerance for the
%  stopping test, KMAX is the maximum number of
%  iterations allowed.
delta=delta0; err=tol+1; k=0; mu=0.1;
eta1=0.25; eta2=0.75; gamma1=0.25; gamma2=2; deltam=5;
xk=x0(:);   gk=grad(xk);  eps2=sqrt(eps);
if meth==1 Hk=hess(xk); else Hk=eye(length(xk)); end
while err>tol & k< kmax
[s]=trustone(Hk,gk,delta);
rho=(fun(xk+s)-fun(xk))/(s'*gk+0.5*s'*Hk*s);
if rho> mu, xk1=xk+s; else, xk1=xk; end
if rho<eta1
     delta=gamma1*delta;
elseif rho> eta2 & abs(norm(s)-delta)<sqrt(eps)
    delta=min([gamma2*delta,deltam]);
end
gk1=grad(xk1);
err=norm((gk1.*xk1)/max([abs(fun(xk1)),1]),inf);
if meth==1 % Newton
   xk=xk1; gk=gk1; Hk=hess(xk);
else       % quasiNewton
  gk1=grad(xk1); yk=gk1-gk; sk=xk1-xk;
  yks=yk'*sk;
  if yks> eps2*norm(sk)*norm(yk)
  Hs=Hk*sk;
  Hk=Hk+(yk*yk')/yks-(Hs*Hs')/(sk'*Hs);
  end
  xk=xk1; gk=gk1;
end
k=k+1;
end
x=xk; iter=k;
if (k==kmax & err > tol)
 fprintf(['The trust region method stopped \n',...
 'without converging to the desired tolerance \n',...
 'because the maximum number of iterations was \n',...
 'reached\n']);
end
```

```
end

function [s]=trustone(Hk,gk,delta)
s=-Hk\gk; d = eigs(Hk,1,'sa');
if norm(s)>delta | d<0
lambda=abs(2*d); I=eye(size(Hk));
for l=1:3
R=chol(Hk+lambda*I);
s=-R \ (R'\gk); q=R'\s;
lambda=lambda+(s'*s)/(q'*q)*(norm(s)-delta)/delta;
if lambda< -d, lambda=abs(lambda*2); end
end
end
end
```

Example 7.11 Let us compute the minimizer of $f(x_1, x_2) = (x_1 + 2x_2 + 2x_1x_2 - 5x_1^2 - 5x_2^2)/(5e^{x_1^2+x_2^2}) + 7/5$ using method (7.59). As seen in Figure 7.12, this function features a local maximum, a saddle point and two local minima, one $\mathbf{x}_{m1}$ in proximity of $(-1., 0.2)$ and the other $\mathbf{x}_{m2}$ in proximity of $(0.3, -0.9)$; the latter is also a global minimizer. We choose $\mathbf{x}^{(0)} = (0, 0.5)$ and compute the matrices H_k recursively, according to (7.48). By calling the Program `trustregion` with the following instructions:

```
fun=@(x)7/5+(x(1)+2*x(2)+2*x(1)*x(2)-5*x(1)^2-...
     5*x(2)^2)/(5*exp(x(1)^2+x(2)^2));
grad_fun=@(x)[(1+2*x(2)-10*x(1)-2*x(1)*(x(1)+2*x(2)+...
2*x(1)*x(2)-5*x(1)^2-5*x(2)^2))/(5*exp(x(1)^2+x(2)^2));
(2+2*x(1)-10*x(2)-2*x(2)*(x(1)+2*x(2)+...
2*x(1)*x(2)-5*x(1)^2-5*x(2)^2))/(5*exp(x(1)^2+x(2)^2))];
delta0=0.5; tol=1.e-5; kmax=100; meth=2; x0=[0;0.5];
[x,err,iter]= trustregion(fun,grad_fun,x0,delta0,...
              tol,kmax,meth)
```

convergence to the point $(.27849, -.89695)$ is achieved in 24 iterations.

Using instead `meth=1` and at each step the exact Hessian matrix, convergence will be achieved in 12 iterations. In both cases a slowing down in convergence is observed when the iterates $\mathbf{x}^{(k)}$ are near the local minimum $\mathbf{x}_{m1}$. The convergence history when using the exact Hessian is reported in Figure 7.12, left, while Figure 7.13, corresponds to using a non-exact Hessian. ■

For the convergence analysis of the trust region method we refer to [NW06, Sez. 4.2] and [SSB85].

The MATLAB command `fminunc` with the `'LargeScale'` option initialized to the value `'on'` implements the trust region method, and the function handle `grad_fun` contains the gradient of the objective function. For instance, using the following instructions:

```
fun=@(x)100*(x(2)-x(1)^2)^2+(1-x(1))^2;  x0=[1.2;-1];
grad_fun=@(x)[-400*(x(2)-x(1)^2)*x(1)-2*(1-x(1));
   200*(x(2)-x(1)^2)];
options = optimset('LargeScale','on','GradObj','on');
[x,fval,exitflag,output]=fminunc({fun,grad_fun},...
          x0,options)
```

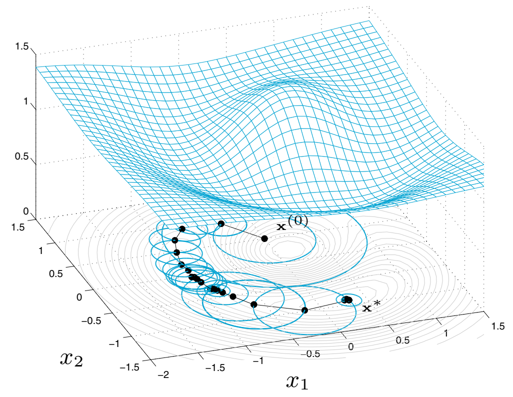


Figure 7.13. Convergence history of the trust region method when H_k are built as in (7.48)

we converge to the minimizer of the Rosenbrock function in 8 iterations; only 9 function evaluations are requested.

Octave 7.2 The `fminunc` command in Octave implements the trust region method with approximated Hessian matrices H_k, computed according to the BFGS recursive formula (7.48). The option `'LargeScale'` is not used in this case. ■

See Exercise 7.7.

7.7 The nonlinear least squares method

In Chapter 3 we have introduced the least squares method for the approximation of either functions or a discrete set of data, by a polynomial (3.29) or another function $\tilde{f}$ linearly depending on a set of unknown coefficients a_j, $j = 1, \ldots, m$. When such a dependence is nonlinear, we face a nonlinear least squares problem.

In abstract terms, let $\mathbf{R}(\mathbf{x}) = (r_1(\mathbf{x}), \ldots, r_n(\mathbf{x}))^T$, with $r_i : \mathbb{R}^m \to \mathbb{R}$, be a given function, and consider the following minimization problem

$$\min_{\mathbf{x}\in\mathbb{R}^m} \Phi(\mathbf{x}) \quad \text{with} \quad \Phi(\mathbf{x}) = \frac{1}{2}\|\mathbf{R}(\mathbf{x})\|^2 = \frac{1}{2}\sum_{i=1}^{n} r_i^2(\mathbf{x}). \qquad (7.61)$$

When the function r_i are nonlinear, Φ might not be convex, featuring several stationary points. All the methods considered thus far, that is Newton's (7.31), descents (7.33) and trust region (7.59), can virtually be used to solve (7.61).

Thanks to the special form of Φ, its gradient and Hessian can be written in terms of the Jacobian $\mathrm{J}_{\mathbf{R}}(\mathbf{x}) \in \mathbb{R}^{n\times m}$ and of the first and second derivatives of $\mathbf{R}$, as follows:

$$\begin{aligned}
&\nabla\Phi(\mathbf{x}) = \mathrm{J}_{\mathbf{R}}(\mathbf{x})^T\mathbf{R}(\mathbf{x}),\\
&\mathrm{H}(\mathbf{x}) = \mathrm{J}_{\mathbf{R}}(\mathbf{x})^T\mathrm{J}_{\mathbf{R}}(\mathbf{x}) + \mathrm{S}(\mathbf{x}),\\
&\text{with } \mathrm{S}_{\ell j}(\mathbf{x}) = \sum_{i=1}^{n}\frac{\partial^2 r_i}{\partial x_\ell \partial x_j}(\mathbf{x})r_i(\mathbf{x}),\ \ \ell, j = 1,\dots,m.
\end{aligned} \tag{7.62}$$

Exact calculation of the Hessian can be cumbersome when m and n are large, especially due to the presence of the matrix $\mathrm{S}(\mathbf{x})$. On the other hand, in several cases the latter matrix is less influent than $\mathrm{J}_{\mathbf{R}}(\mathbf{x})^T\mathrm{J}_{\mathbf{R}}(\mathbf{x})$ and could be approximated or even neglected in the construction of $\mathrm{H}(\mathbf{x})$. This is the case of the two methods that we are going to present in the next two sections.

7.7.1 Gauss-Newton method

This method is a variant of the Newton method (7.31) for the solution of (7.61) in which the exact Hessian $\mathrm{H}(\mathbf{x})$ is approximated by neglecting $\mathrm{S}(\mathbf{x})$ in (7.62).

Its formulation is as follows: given $\mathbf{x}^{(0)} \in \mathbb{R}^m$, for $k = 0, 1, \dots,$ until convergence:

$$\boxed{\begin{aligned}
&\text{solve } \left[\mathrm{J}_{\mathbf{R}}(\mathbf{x}^{(k)})^T\mathrm{J}_{\mathbf{R}}(\mathbf{x}^{(k)})\right]\boldsymbol{\delta}\mathbf{x}^{(k)} = -\mathrm{J}_{\mathbf{R}}(\mathbf{x}^{(k)})^T\mathbf{R}(\mathbf{x}^{(k)})\\
&\text{set } \mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \boldsymbol{\delta}\mathbf{x}^{(k)}
\end{aligned}} \tag{7.63}$$

If $\mathrm{J}_{\mathbf{R}}(\mathbf{x}^{(k)})$ has not full rank, the linear system $(7.63)_1$ features infinitely many solutions, in which case the Gauss-Newton method can stagnate, diverge, or converge to a non-stationary point.

If instead $\mathrm{J}_{\mathbf{R}}(\mathbf{x}^{(k)})$ has full rank, system $(7.63)_1$ features the form (5.42) and can be solved using either a QR factorization or a singular value decomposition of $\mathrm{J}_{\mathbf{R}}(\mathbf{x}^{(k)})$ as seen in Section 5.7.

It can be proved (see Exercise 7.8) that neglecting $\mathrm{S}(\mathbf{x}^{(k)})$ at the step k of the minimization process amounts to approximate $\mathbf{R}(\mathbf{x})$ with its Taylor development centered at $\mathbf{x}^{(k)}$ and truncated at the first order

$$\widetilde{\mathbf{R}}_k(\mathbf{x}) = \mathbf{R}(\mathbf{x}^{(k)}) + \mathrm{J}_{\mathbf{R}}(\mathbf{x}^{(k)})(\mathbf{x} - \mathbf{x}^{(k)}). \tag{7.64}$$

The convergence of Gauss-Newton method is not always guaranteed. It actually depends on both the property of Φ and the choice of the initial point. The following result is proved in [JS96]: if $\mathbf{x}^*$ is a stationary point for Φ and $\mathrm{J}_{\mathbf{R}}(\mathbf{x})$ has full rank in a suitable neighborhood of $\mathbf{x}^*$, we have:

1. if $\mathrm{S}(\mathbf{x}^*) = 0$, which is the case if $\mathbf{R}(\mathbf{x})$ is linear or $\mathbf{R}(\mathbf{x}^*) = \mathbf{0}$, the Gauss-Newton method is locally (quadratically) convergent (in fact it coincides with Newton's method);
2. if $\|\mathrm{S}(\mathbf{x}^*)\|_2$ is small with respect to the minimum (positive) eigenvalue of $\mathrm{J}_{\mathbf{R}}(\mathbf{x}^*)^T\mathrm{J}_{\mathbf{R}}(\mathbf{x}^*)$, then Gauss-Newton method converges linearly. This is for instance the case if $\mathbf{R}(\mathbf{x})$ is nonlinear with a small non-linearity or if $\mathbf{R}(\mathbf{x}^*)$ is small;
3. if $\|\mathrm{S}(\mathbf{x}^*)\|_2$ is large with respect to the minimum (positive) eigenvalue of $\mathrm{J}_{\mathbf{R}}(\mathbf{x}^*)^T\mathrm{J}_{\mathbf{R}}(\mathbf{x}^*)$, the Gauss-Newton method might not converge even if $\mathbf{x}^{(0)}$ is very close to $\mathbf{x}^*$. This happens if $\mathbf{R}(\mathbf{x})$ is strongly nonlinear or its residual $\mathbf{R}(\mathbf{x}^*)$ is large.

Remark 7.5 Line search techniques can be used in combination with the Gauss-Newton method by replacing $(7.63)_2$ with $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k\boldsymbol{\delta}\mathbf{x}^{(k)}$, where the computation of the steplengths α_k is described in Section 7.5.1. If $\mathrm{J}_{\mathbf{R}}(\mathbf{x}^{(k)})$ has full rank, the matrix $\mathrm{J}_{\mathbf{R}}(\mathbf{x}^{(k)})^T\mathrm{J}_{\mathbf{R}}(\mathbf{x}^{(k)})$ is symmetric and positive definite and $\boldsymbol{\delta}\mathbf{x}^{(k)}$ is a descent direction for Φ (see Exercise 7.9). In this case, under suitable assumptions on Φ, we obtain a globally convergent method, called *damped Gauss-Newton* method. ■

The Gauss-Newton method is implemented in Program 7.5; `r` and `jr` are function handles associated with the function $\mathbf{R}(\mathbf{x})$ and its Jacobian $\mathrm{J}_{\mathbf{R}}(\mathbf{x})$, respectively, `x0` is the initial vector, while `tol` and `kmax` contain the tolerance for the stopping test and the maximum number of iterations allowed. The output variable `x` contains the computed solution, `err` an estimate of the error at the last iteration and `iter` the number of iterations required to converge.

Program 7.5. gaussnewton: Gauss-Newton method

```
function [x,err,iter]= gaussnewton(r,jr,x0,tol,...
    kmax,varargin)
%GAUSSNEWTON  Solves nonlinear least squares problems
%  [X,ERR,ITER]=GAUSSNEWTON(R,JR,X0,TOL,KMAX)
%  solves the nonlinear least squares by the Gauss-
%  Newton method. R and JR are the function handles
%  associated with the function R and its Jacobian,
%  respectively. X0 is the initial point for the se-
%  quence. TOL is the tolerance for the stopping test,
%  KMAX is the maximum number of allowed iterations.
err=tol+1; k=0; xk=x0(:);
rk=r(xk,varargin{:}); jrk=jr(xk,varargin{:});
while err>tol & k< kmax
[Q,R]=qr(jrk,0); dk=-R \ (Q'*rk);
xk1=xk+dk;
rk1=r(xk1,varargin{:});
jrk1=jr(xk1,varargin{:});
k=k+1;  err=norm(xk1-xk);
xk=xk1; rk=rk1; jrk=jrk1;
end
x=xk; iter=k;
if (k==kmax & err > tol)
```

```
 fprintf(['Gauss-Newton method stopped \n',...
 'without converging to the desired tolerance \n',...
 'because the maximum number of iterations was \n',...
 'reached\n']);
end
```

Example 7.12 Let us consider Problem 7.2 under the form (7.5) (a special case of (7.61)). We use the Gauss-Newton method (7.63), we storage vector $\mathbf{a}$ in the upper part of $\mathbf{x}$ and $\boldsymbol{\sigma}$ in the lower one, yielding

$$r_i(\mathbf{x}) = f(t_i;\mathbf{a},\boldsymbol{\sigma}) - y_i = \sum_{k=1}^{m} f_k(t_i;a_k,\sigma_k) - y_i,$$

$$\frac{\partial r_i}{\partial a_k} = f_k(t_i;a_k,\sigma_k)\frac{t_i-a_k}{\sigma_k^2}, \qquad \frac{\partial r_i}{\partial \sigma_k} = f_k(t_i;a_k,\sigma_k)\left[\frac{(t_i-a_k)^2}{\sigma_k^3} - \frac{1}{2\sigma_k}\right].$$

We generate the n points (t_i, y_i) with $i = 1,\dots,n$, $0 \le t_i \le 10$, by summing 5 Gaussian functions of the form (7.3) taking $\mathbf{a} = [2.3, 3.25, 4.82, 5.3, 6.6]$, $\boldsymbol{\sigma} = [0.2, 0.34, 0.50, 0.23, 0.39]$ and adding a random noise:

```
a=[2.3,3.25,4.82,5.3,6.6]; m=length(a);
sigma=[0.2,0.34,0.50,0.23,0.39];
gaussian=@(t,a,sigma)...
  exp(-((t-a)/(sqrt(2)*sigma)).^2)/(sqrt(pi*2)*sigma);
n=2000; t=linspace(0,10,n)'; y=zeros(n,1);
for k=1:m, y=y+gaussian(t,a(k),sigma(k)); end
y=y+0.05*randn(n,1);
```

We now call Program 7.5 using the following instructions:

```
x0=[2,3,4,5,6,0.3,0.3,0.6,0.3,0.3];
tol=3.e-5; kmax=200;
[x,err,iter]=gaussnewton(@gmmr,@gmmjr,x0,tol,kmax,t,y)
xa=x(1:m); xsigma=x(m+1:end);
h=1./(sqrt(2*pi)*xsigma); w=2*sqrt(log(4))*xsigma;
```

where gmmr and gmmjr are the functions defining $\mathbf{R}(\mathbf{x})$ and $\mathrm{J}_{\mathbf{R}}(\mathbf{x})$, respectively.

```
function [R]=gmmr(x,t,y)
x=x(:);
m=length(x)/2; a=x(1:m); sigma=x(m+1:end);
n=length(t); R=zeros(n,1);
gaussian=@(t,a,sigma)[exp(-((t-a)/(sqrt(2)*sigma))...
     .^2)/(sqrt(pi*2)*sigma)];
for k=1:m, R=R+gaussian(t,a(k),sigma(k)); end, R=R-y;

function [Jr]=gmmjr(x,t,y)
x=x(:); m=length(x)/2; a=x(1:m); sigma=x(m+1:end);
n=length(t); Jr=zeros(n,m*2);
gaussian=@(t,a,sigma)[exp(-((t-a)/(sqrt(2)*sigma))...
    .^2)/(sqrt(pi*2)*sigma)];
fk=zeros(n,m);
for k=1:m, fk(:,k)=gaussian(t,a(k),sigma(k)); end
for k=1:m, Jr(:,k)=(fk(:,k).*(t-a(k))/sigma(k)^2)'; end
for k=1:m, Jr(:,k+m)=(fk(:,k).*((t-a(k)).^2/...
          sigma(k)^3-1/(2*sigma(k))))'; end
```

Convergence is achieved in 22 iterations. The vectors `xa` and `xsigma` contain the approximation of vectors $\mathbf{a}$ and $\boldsymbol{\sigma}$, respectively, while `h` and `w` contain the

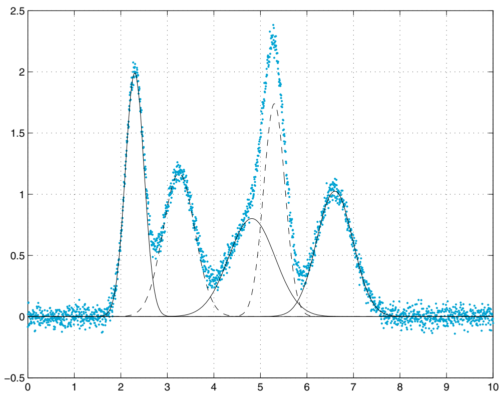


Figure 7.14. I dati (in azzurro) e la soluzione (in nero) dell'Esempio 7.12

height and amplitude, respectively, of the Gaussian functions we are looking for.

We display in Figure 7.14 the points (t_i, y_i) (in blue) representing the signal and the 5 Gaussian functions (7.3) (black lines) built on the obtained numerical solution. This is the case with large residual: as a matter of fact $\Phi(\mathbf{x}^*) = 1.0385e + 03$, $\mathbf{x}^*$ being the solution vector. By a slight change of the initial data, for instance by simply modifying the last component of $\mathbf{x}^{(0)}$ from 0.3 to 0.5, the method would not converge any more. This remark prompts us to a convenient choice of $\mathbf{x}^{(0)}$. ■

7.7.2 Levenberg-Marquardt's method

This is a trust region method for the solution of the minimization problem (7.61). Following algorithm (7.59), after replacing f with Φ (see (7.61)) and $\tilde{f}$ with $\tilde{\Phi}$, at each step k we solve the minimization problem

$$\min_{\mathbf{s}\in\mathbb{R}^n:\ \|\mathbf{s}\|\le\delta_k} \tilde{\Phi}_k(\mathbf{s})$$

with

$$\tilde{\Phi}_k(\mathbf{s}) = \frac{1}{2}\|\mathbf{R}(\mathbf{x}^{(k)}) + \mathrm{J}_{\mathbf{R}}(\mathbf{x}^{(k)})\mathbf{s}\|^2. \tag{7.65}$$

Note that $\tilde{\Phi}_k(\mathbf{x})$ (7.65) is a quadratic approximation of $\Phi(\mathbf{x})$ around $\mathbf{x}^{(k)}$, obtained by approximating $\mathbf{R}(\mathbf{x})$ with its linear model $\widetilde{\mathbf{R}}_k(\mathbf{x})$ (7.64) (see Exercise 7.11).

Even though $J_{\mathbf{R}}(\mathbf{x})$ does not have full rank, this method is well suited for minimization problems featuring a strong non-linearity or a large residual $\Phi(\mathbf{x}^*) = \frac{1}{2}\|\mathbf{R}(\mathbf{x}^*)\|^2$ in correspondence with a local minimizer $\mathbf{x}^*$.

Since the approximation of the Hessian matrix is the same as for the Gauss-Newton method, the two methods share the same local convergence properties. In particular, should the Levenberg-Marquardt iterations converge, convergence rate is quadratic if the residual is null at local minimizer, linear otherwise.

See Exercises 7.8-7.11.

Let us summarize

1. For the minimization of the function f, the derivative free methods are those using only the functional values of f. They are quite robust in practice even though very little is known about their theoretical convergence;
2. descent methods exploit the knowledge of the function derivatives and compute at each step a descent direction and a steplength, based on line search strategies;
3. descent methods with Newton directions associated with linear search strategies are globally convergent when the matrices $\mathrm{H}(\mathbf{x}^{(k)})$ are positive definite. They feature quadratic convergence rate in proximity of the minimizer. They are well suited for small and medium size problems;
4. descent methods with quasi-Newton directions make use of approximate Hessian matrices H_k at every iteration. When associated with line search strategies, they are globally convergent provided H_k are positive definite, with superlinear convergence order. They too are well suited for small and medium size problems;
5. descent methods with conjugate gradient type descent directions, associated with line search strategies, are globally convergent with linear rate of convergence. They are recommended for large size problems;
6. trust region strategies are more recent and less diffused than line search ones. They replace the objective function with a quadratic approximation and look for a minimizer of the latter in a n-dimensional ball.

7.8 Constrained optimization

In this Section we introduce two simple strategies for the solution of minimization problems with constraints: the penalty method for problems with both equality and inequality constraints and the so-called augmented Lagrangian method for problems featuring equality constraints only.

These two methods allow the solutions of simple problems and provide the basic tools for more robust and complex algorithms that we will not address here (see however [NW06, SY06, BDF+10]).

The constrained optimization problem is formulated as follows: we consider the minimization problem (7.2) for which the domain Ω can be either given by

$$\Omega = \{\mathbf{x} \in \mathbb{R}^n : \ h_i(\mathbf{x}) = 0, \ \text{for } i = 1, \ldots, p\}, \tag{7.66}$$

where $h_i : \mathbb{R}^n \to \mathbb{R}$ for $i = 1, \ldots, p$, are given functions, or by

$$\Omega = \{\mathbf{x} \in \mathbb{R}^n : \ g_j(\mathbf{x}) \geq 0, \ \text{for } j = 1, \ldots, q\}, \tag{7.67}$$

where $g_j : \mathbb{R}^n \to \mathbb{R}$ for $j = 1, \ldots, q$; p and q are given natural numbers. In the more general case, however, Ω is defined by both equality and inequality constraints, that is

$$\Omega = \{\mathbf{x} \in \mathbb{R}^n : h_i(\mathbf{x}) = 0, \ \text{for } i = 1, \ldots, p, \ g_j(\mathbf{x}) \geq 0, \ \text{for } j = 1, \ldots, q\}. \tag{7.68}$$

The three different situations (7.66), (7.67), and (7.68) undergo a unique notation,

$$\Omega = \{\mathbf{x} \in \mathbb{R}^n : h_i(\mathbf{x}) = 0, \ \text{for } i \in \mathcal{I}_h, \ g_j(\mathbf{x}) \geq 0, \ \text{for } j \in \mathcal{I}_g\},$$

for two suitable chosen sets $\mathcal{I}_h$ and $\mathcal{I}_g$, under the convention that $\mathcal{I}_h = \emptyset$ in (7.67) and $\mathcal{I}_g = \emptyset$ in (7.66).

Problem (7.2) can thus be written as

$$\boxed{\begin{array}{l} \min_{\mathbf{x} \in \mathbb{R}^n} f(\mathbf{x}), \ \text{subject to} \\ h_i(\mathbf{x}) = 0 \quad \forall i \in \mathcal{I}_h, \\ g_j(\mathbf{x}) \geq 0 \quad \forall j \in \mathcal{I}_g \end{array}} \tag{7.69}$$

Everywhere in this section we will assume that f, h_i, and g_j be C^1 functions on $\mathbb{R}^n$.

The points of $\mathbf{x} \in \Omega$ are called *admissibile* (as they fulfill all the constraints); Ω is the set of admissible points.

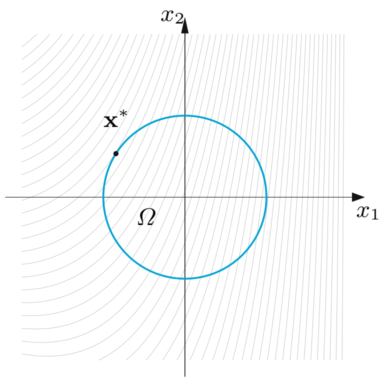
A point $\mathbf{x}^* \in \Omega \subset \mathbb{R}^n$ is a *global minimizer* for problem (7.2) if

$$f(\mathbf{x}^*) \leq f(\mathbf{x}) \qquad \forall \mathbf{x} \in \Omega,$$

whereas $\mathbf{x}^*$ is a *local minimizer* for (7.2) if there exists a ball $B_r(\mathbf{x}^*) \subset \mathbb{R}^n$ with radius $r > 0$ and centered at $\mathbf{x}^*$ such that

$$f(\mathbf{x}^*) \leq f(\mathbf{x}) \qquad \forall \mathbf{x} \in B_r(\mathbf{x}^*) \cap \Omega.$$

A constraint is said *active* at $\mathbf{x} \in \Omega$ if it is satisfied with equality at $\mathbf{x} \in \Omega$. According to this definition, active constraints at $\mathbf{x}$ are all the h_i as well as those g_j such that $g_j(\mathbf{x}) = 0$.



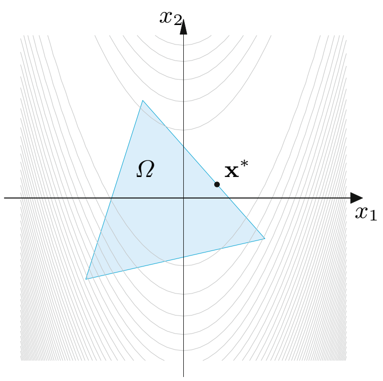


Figure 7.15. The contour lines of the cost function f, the admissibility set Ω and the global minimizer $\mathbf{x}^*$ constrained to Ω. The plot at left is relative to Problem 1 (7.70), that at right to Problem 2 (7.71)

Example 7.13 Consider the following constrained optimization problems: *Problem 1:*

$$\min_{\mathbf{x}\in\mathbb{R}^2} f(\mathbf{x}), \qquad \text{with } f(\mathbf{x}) = \frac{3}{5}x_1^2 + \frac{1}{2}x_1x_2 - x_2 + 3x_1,$$
$$\text{under the following constraint} \tag{7.70}$$
$$h_1(\mathbf{x}) = x_1^2 + x_2^2 - 1 = 0;$$

Problem 2:

$$\min_{\mathbf{x}\in\mathbb{R}^2} f(\mathbf{x}), \qquad \text{with } f(\mathbf{x}) = 100(x_2 - x_1^2)^2 + (1-x_1)^2,$$
$$\text{under the following constraints:}$$
$$g_1(\mathbf{x}) = -34x_1 - 30x_2 + 19 \geq 0, \tag{7.71}$$
$$g_2(\mathbf{x}) = 10x_1 - 5x_2 + 11 \geq 0,$$
$$g_3(\mathbf{x}) = 3x_1 + 22x_2 + 8 \geq 0.$$

The contour lines of the two cost functions and the associated set of admissible points Ω are displayed in Figure 7.15. Note that Ω is a closed curve for Problem 1, while it is a closed convex set in $\mathbb{R}^2$ for Problem 2. For both problems there is one active constraint. ■

The Weierstrass theorem guarantees the existence of both the maximum and the minimum for f in Ω when the latter is a non-empty, bounded and closed set. Consequently, problem (7.69) admits a solution.

We recall that a function $f : \Omega \subseteq \mathbb{R}^n \to \mathbb{R}$ is *strongly convex* in Ω if $\exists\rho > 0$ such that $\forall\mathbf{x}, \mathbf{y} \in \Omega$ and $\forall\alpha \in [0, 1]$,

$$f(\alpha\mathbf{x} + (1-\alpha)\mathbf{y}) \leq \alpha f(\mathbf{x}) + (1-\alpha)f(\mathbf{y}) - \alpha(1-\alpha)\rho\|\mathbf{x}-\mathbf{y}\|^2. \tag{7.72}$$

This reduces to the definition of convexity (7.11) when $\rho = 0$.

Proposition 7.2 (Optimality conditions) *Let $\Omega \subset \mathbb{R}^n$ be a convex set, and $\mathbf{x}^* \in \Omega$ be such that $f \in C^1(B_r(\mathbf{x}^*))$ for a suitable $r > 0$. If $\mathbf{x}^*$ is a local minimizer for (7.2) then*

$$\nabla f(\mathbf{x}^*)^T(\mathbf{x} - \mathbf{x}^*) \geq 0 \qquad \forall \mathbf{x} \in \Omega. \tag{7.73}$$

Moreover, if f is convex in Ω and (7.73) is satisfied, $\mathbf{x}^*$ is a global minimizer for (7.2).
Finally, under the additional requirement for Ω to be closed and f strongly convex, the minimizer for (7.2) is unique.

Let us introduce the *Lagrangian function* associated with problem (7.2)

$$\mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) = f(\mathbf{x}) - \sum_{i \in \mathcal{I}_h} \lambda_i h_i(\mathbf{x}) - \sum_{j \in \mathcal{I}_g} \mu_j g_j(\mathbf{x}). \tag{7.74}$$

Here $\boldsymbol{\lambda} = (\lambda_i)$ (for $i \in \mathcal{I}_h$) and $\boldsymbol{\mu} = (\mu_j)$ (for $j \in \mathcal{I}_g$) play the role of *Lagrangian multipliers* associated with equality and inequality constraints, respectively. A point $\mathbf{x}^*$ is called a *Karush–Kuhn–Tucker* (KKT) point for $\mathcal{L}$ if there exist $\boldsymbol{\lambda}^*$ and $\boldsymbol{\mu}^*$ such that the triplet $(\mathbf{x}^*, \boldsymbol{\lambda}^*, \boldsymbol{\mu}^*)$ satisfies the following conditions, called *Karush–Kuhn–Tucker conditions*:

$$\begin{aligned}
&\nabla_{\mathbf{x}}\mathcal{L}(\mathbf{x}^*, \boldsymbol{\lambda}^*, \boldsymbol{\mu}^*) = \nabla f(\mathbf{x}^*) - \sum_{i \in \mathcal{I}_h} \lambda_i^* \nabla h_i(\mathbf{x}^*) - \sum_{j \in \mathcal{I}_g} \mu_j^* \nabla g_j(\mathbf{x}^*) = \mathbf{0}\\
&h_i(\mathbf{x}^*) = 0 \quad \forall i \in \mathcal{I}_h\\
&g_j(\mathbf{x}^*) \geq 0 \quad \forall j \in \mathcal{I}_g\\
&\mu_j^* \geq 0 \quad \forall j \in \mathcal{I}_g\\
&\mu_j^* g_j(\mathbf{x}^*) = 0 \quad \forall j \in \mathcal{I}_g
\end{aligned}$$

For a given point $\mathbf{x}$, the constraints are said to satisfy the LICQ (*linear independence constraint qualification*) condition at $\mathbf{x}$ if the gradients $\nabla h_i(\mathbf{x})$ and $\nabla g_j(\mathbf{x})$ associated with the sole active constraints at $\mathbf{x}$ provide a set of linear independent vectors.

The following result holds (see, e.g., [NW06, Thm. 12.1]).

Theorem 7.1 (First order KKT necessary conditions) *If* $\mathbf{x}^*$ *is a local minimizer for problem* (7.69)*, the functions* f*,* h_i*, and* g_j *are of class* $C^1(\Omega)$*, and the constraints satisfy the LICQ condition at* $\mathbf{x}^*$*, then there exist* $\boldsymbol{\lambda}^*$ *and* $\boldsymbol{\mu}^*$ *such that* $(\mathbf{x}^*, \boldsymbol{\lambda}^*, \boldsymbol{\mu}^*)$ *is a KKT point.*

Thanks to this theorem, the local minimizers for (7.69) should be sought for among the KKT points and those points where LICQ condition is not fulfilled.

When the set $\mathcal{I}_g$ is empty (only equality constraints are present) the Lagrangian function reads $\mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}) = f(\mathbf{x}) - \sum_{i\in\mathcal{I}_h} \lambda_i h_i(\mathbf{x})$ and the KKT conditions reduce to the classical necessary (*Lagrangian*) conditions

$$\boxed{\begin{aligned} &\nabla_{\mathbf{x}}\mathcal{L}(\mathbf{x}^*, \boldsymbol{\lambda}^*) = \nabla f(\mathbf{x}^*) - \sum_{i\in\mathcal{I}_h} \lambda_i^* \nabla h_i(\mathbf{x}^*) = \mathbf{0} \\ &h_i(\mathbf{x}^*) = 0 \quad \forall i \in \mathcal{I}_h \end{aligned}} \tag{7.75}$$

Sufficient conditions for a KKT point to be a minimizer for f constrained in Ω would require the knowledge of the Hessian matrix of $\mathcal{L}$ or else an assumption of strict convexity for both f and the constraint functions ([NW06, SY06]).

In general terms, a constrained optimization problem can be written as an unconstrained problem using either the penalized formulation or the augmented Lagrangian formulation, as we will explain in the next two sections.

Remark 7.6 If at a point $\mathbf{x}^*$ that minimizes f no active constraints are present, the Lagrangian function coincides with the cost function f therein, as $\mathcal{I}_h = \emptyset$ and $\mu_j^* = 0$ for all $j \in \mathcal{I}_g$ thanks to the KKT conditions. In this case our problem reduces to an unconstrained minimization problem that can be solved by using the methods discussed in the previous sections. ■

A remarkable instance of constrained optimization is that of *Quadratic Programming*: this is precisely the case where f is a quadratic function, the constraints are expressed by linear functions, thus problem (7.69) can be written under the special form:

$$\boxed{\begin{array}{ll} \min\limits_{\mathbf{x}\in\mathbb{R}^n} f(\mathbf{x}), & f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T \mathrm{A}\mathbf{x} + \mathbf{x}^T\mathbf{b} \\ \text{subject to the constraints} & \mathrm{C}\mathbf{x} - \mathbf{d} = \mathbf{0}, \quad \mathrm{D}\mathbf{x} - \mathbf{e} \geq \mathbf{0} \end{array}} \tag{7.76}$$

where $\mathrm{A} \in \mathbb{R}^{n\times n}$, $\mathbf{b} \in \mathbb{R}^n$, $\mathrm{C} \in \mathbb{R}^{p\times n}$, $\mathbf{d} \in \mathbb{R}^p$, $\mathrm{D} \in \mathbb{R}^{q\times n}$, $\mathbf{e} \in \mathbb{R}^q$, p, q are suitable positive integers and the notations $\mathbf{v} \geq \mathbf{0}$ means $v_i \geq 0$ for all i. See [Bom10, NW06] for a presentation of Quadratic Programming.

In the special case where constraints are all expressed by equalities, the matrix form of the Langrange conditions (7.75) reads (with obvious choice of notations)

$$\begin{bmatrix} \mathrm{A} & -\mathrm{C}^T \\ \mathrm{C} & 0 \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ \boldsymbol{\lambda} \end{bmatrix} = \begin{bmatrix} -\mathbf{b} \\ \mathbf{d} \end{bmatrix}. \tag{7.77}$$

If A is symmetric and positive definite on the kernel of C, that is

$$\mathbf{y}^T\mathrm{A}\mathbf{y} > 0 \quad \forall \mathbf{y} \in \mathit{ker}(\mathrm{C}) = \{\mathbf{z} : \mathrm{C}\mathbf{z} = \mathbf{0}\},\ \mathbf{y} \neq \mathbf{0},$$

and assuming that C has full rank, system (7.77) admits a unique solution, thus there exists a unique global minimizer for the cost function defined in (7.76).

A quadratic programming problem can therefore be tackled by solving the linear system (7.77) using one of the methods of Chapter 5.

In general, the matrix $\mathrm{M} = [\mathrm{A},\ -\mathrm{C}^T; \mathrm{C},\ 0]$ of (7.77) is not definite, that is it features both positive and negative eigenvalues. Suitable iterative methods for its treatment are Krylov methods like GMRES or Bi-CGStab. See, e.g., [Qua13] and [BGL05].

Example 7.14 To solve Problem 7.4 we note that the cost function defined in (7.7) (the risk) is quadratic, while the constraints read

$$h_1(\mathbf{x}) = 0.6x_1 + x_2 + 1.2x_3 = 1.04, \quad h_2(\mathbf{x}) = x_1 + x_2 + x_3 = 1. \tag{7.78}$$

The former states that the expected return be equal to 10.4%, while the latter establishes that the sum of the fractions invested into the 3 funds be equal to the entire capital. This is a quadratic programming problem that we can rewrite under the form (7.77), where

$$\mathrm{A} = \begin{bmatrix} 0.08 & 0.1 & 0 \\ 0.1 & 0.5 & 0.208 \\ 0 & 0.208 & 1.28 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad \mathrm{C} = \begin{bmatrix} 0.6 & 1 & 1.2 \\ 1 & 1 & 1 \end{bmatrix}, \quad \mathbf{d} = \begin{bmatrix} 1.04 \\ 1 \end{bmatrix}.$$

Matrix C has (maximum) rank equal to 2, its kernel $\mathit{ker}(\mathrm{C}) = \{\mathbf{z} = \alpha[1, -3, 2]^T, \alpha \in \mathbb{R}\}$ has dimension 1.

As A is symmetric, we need to verify that it is positive definite on $\mathit{ker}(\mathrm{C})$, that is $\mathbf{z}^T\mathrm{A}\mathbf{z} > 0$ for all $\mathbf{z} = \alpha[1, -3, 2]^T$, $\alpha \neq 0$. As a matter of fact, $\mathbf{z}^T\mathrm{A}\mathbf{z} = \alpha^2[1, -3, 2]^T\mathrm{A}[1, -3, 2] = 6.6040\alpha^2 > 0$. Upon building the matrix $\mathrm{M} = [\mathrm{A},\ -\mathrm{C}^T; \mathrm{C},\ 0]$ and the right hand side $\mathbf{f} = [-\mathbf{b}, \mathbf{d}]^T$, we solve (7.77) using the following instructions:

```
A=[0.08 0.1 0; 0.1 0.5 0.208; 0 0.208 1.28]; b=[0;0;0];
C=[0.6 1 1.2;1,1,1]; d=[1.04;1];
M=[A -C'; C, zeros(2)]; f=[-b;d];
xl=M\f
```

and obtain the solution

```
xl =
    0.0606
    0.6183
    0.3211
    0.7883
   -0.4063
```

The first 3 components of `xl` correspond to the 3 fractions of the capital to invest in the 3 funds, whereas the last two components provide the values of the Lagrangian multipliers associated with the constraints. The risk corresponding to this capital splitting is given by the value of the cost function at the point `xl(1:3)` and is approximately equal to 21%. ■

7.8.1 The penalty method

A strategy for solving problem (7.69) consists of turning it into a non-constrained optimization problem for a modified *penalty function*

$$\mathcal{P}_\alpha(\mathbf{x}) = f(\mathbf{x}) + \frac{\alpha}{2}\sum_{i\in\mathcal{I}_h} h_i^2(\mathbf{x}) + \frac{\alpha}{2}\sum_{j\in\mathcal{I}_g}(\max\{-g_j(\mathbf{x}),0\})^2 \quad (7.79)$$

where $\alpha > 0$ is a parameter to be chosen.

If the given constraints are not fulfilled at the point $\mathbf{x}$, the sums appearing in (7.79) provide a measure of how far $\mathbf{x}$ is from the admissible set Ω. Since in this case $\mathbf{x}$ violates the constraints, choosing large values of α would severely penalize such a violation. Every solution $\mathbf{x}^*$ of (7.69) clearly provides a minimizer of $\mathcal{P}$. Conversely, assuming f, h_i, and g_j regular enough, and denoting with $\mathbf{x}^*(\alpha)$ a minimizer of $\mathcal{P}_\alpha(\mathbf{x})$, it holds ([Ber82])

$$\lim_{\alpha\to\infty} \mathbf{x}^*(\alpha) = \mathbf{x}^*.$$

For $\alpha \gg 1$, $\mathbf{x}^*(\alpha)$ can therefore be regarded as a convenient approximation of $\mathbf{x}^*$. However, since numerical instabilities arising from the minimization of $\mathcal{P}_\alpha(\mathbf{x})$ increase with α, a better strategy consists of solving a sequence of unconstrained minimization problems

$$\mathbf{x}^{(k)} = \underset{\mathbf{x}\in\mathbb{R}^n}{\operatorname{argmin}}\, \mathcal{P}_{\alpha_k}(\mathbf{x}) \quad (7.80)$$

where $\{\alpha_k\}$ is a monotonically increasing unbounded sequence of parameters (with, e.g., $\alpha_0 = 1$). For every k, α_{k+1} is chosen as a function of α_k and $\mathbf{x}^{(k)}$ provides the initial value for problem (7.80) at the new step $k+1$.

A heuristic approach consists of choosing $\alpha_{k+1} = \delta\alpha_k$ where δ is small (say $\delta \in [1.5, 2]$) if many iterations have been necessary to solve (7.80) at the step k, otherwise one could afford a larger value for δ, say $\delta \simeq 10$.

As a matter of fact, in the course of the first iterations, when using a moderate (not too high) α_k, there is no reason why the solution of (7.80) should resemble that of (7.69). This legitimates the search for an inexact solution of (7.80), differing from the exact one $\mathbf{x}^{(k)}$ by a small enough tolerance ε_k.

The algorithm above is formulated as follows (note that a further tolerance $\overline{\varepsilon} > 0$ is requested to assess the behaviour of the gradient of $\mathcal{P}$ at $\mathbf{x}^{(k)}$).

For given α_0 (tipically, $\alpha_0 = 1$), ε_0 (tipically, $\varepsilon_0 = 1/10$), $\overline{\varepsilon} > 0$ and $\mathbf{x}_0^{(0)} \in \mathbb{R}^n$, for $k = 0, 1, \ldots$ until convergence

```
compute an approximation $\mathbf{x}^{(k)}$ to (7.80) using an initial
data $\mathbf{x}_0^{(k)}$ and a tolerance $\varepsilon_k$ on the stopping criterium;
if $\|\nabla_{\mathbf{x}}\mathcal{P}_{\alpha_k}(\mathbf{x}^{(k)})\| \leq \overline{\varepsilon}$
    set $\mathbf{x}^* = \mathbf{x}^{(k)}$ (convergence achieved)
else                                                              (7.81)
    choose $\alpha_{k+1}$ s.t. $\alpha_{k+1} > \alpha_k$
    choose $\varepsilon_{k+1}$ s.t. $\varepsilon_{k+1} < \varepsilon_k$
    set $\mathbf{x}_0^{(k+1)} = \mathbf{x}^{(k)}$
endif
```

This alogorithm is implemented in Program 7.6. `fun` and `grad_fun` are function handles associated with the cost function and its gradient, respectively; `h` and `grad_h` are those associated with the equality constraint functions, while `g` and `grad_g` those associated with inequality constraint functions. When $\mathcal{I}_h$ (resp., $\mathcal{I}_g$) is an empty set, `h` and `grad_h` (resp. `g` and `grad_g`) are empty variables. The output of the functions `grad_fun`, `grad_h` and `grad_g` respectively contain: an n dimensional column vector $\mathbf{y}$ with components $y_i = \partial f/\partial x_i$, an $n \times p$ matrix C whose coefficients are $\mathrm{C}_{ji} = \partial h_i/\partial x_j$, an $n \times q$ matrix G whose entries are $\mathrm{G}_{j\ell} = \partial g_\ell/\partial x_j$. The vector `x0` contains $\mathbf{x}_0^{(0)}$, `tol` and `kmax` the tolerance and the maximum number of iterations for the penalty loop, while `kmaxd` is the maximum number of iterations for the descent method, when the latter is called at every step to solve the unconstrained minimization problem. In this program the tolerance ε_k for the descent method is chosen equal to $1/10$ for $k = 0$ and then reduced at every iteration by a factor 10 until the tolerance $\overline{\varepsilon}$ is reached. The variable `meth` is used to select the unconstrained minimization method: if `meth=0` the MATLAB `fminsearch` function implementing the Nelder and Mead method is chosen, while `meth>1` has the same role played in Program `descent`

7.3 to select the descent method. Finally, if `meth=1`, the Hessian matrix necessary to implement the descent method with Newton's directions is provided as input variable, while it provides H_0 for the BFGS method (7.49) if `meth=2`.

Program 7.6. penalty: penalty method

```
function [x,err,k]=penalty(fun,grad_fun,h,grad_h,...
g,grad_g,x0,tol,kmax,kmaxd,meth,varargin)
% PENALTY Constrained  optimization with penalty
%  [X,ERR,K]=PENALTY(FUN,GRAD_FUN,H,GRAD_H,...
%  G,GRAD_G,X0,TOL,KMAX,KMAXD,METH)
%  computes a local minimizer of the cost function
%  FUN under the constraints H=0 and G>=0, by the
%  penalty method. X0 is the initial point, TOL is
%  the tolerance for the stopping test, KMAX is the
%  maximum number of allowed iterations.
%  GRAD_FUN, GRAD_H, and GRAD_G contain the gradient
%  of FUN, H, and G, respectively. The variables
%  H, G, GRAD_H, and GRAD_G can be set to [], if they
%  are not present. The solution of the corresponding
%  unconstrained minimization problem is performed
%  by calling either Matlab FMINSEARCH function
%  (if METH=0) or DESCENT function (if METH>0).
%  When METH>0, KMAXD and METH contain respectively
%  the maximum number of allowed iterations for the
%  function DESCENT and the choice of the descent
%  directions. When METH>1
%  [X,ERR,K]=PENALTY(FUN,GRAD_FUN,H,GRAD_H,...
%  G,GRAD_G,X0,TOL,KMAX,KMAXD,METH, HESS)
%  is the correct calling instruction.
%  If METH=1 HESS is the function handle associated
%  with the Hessian is required, if METH=2 HESS is a
%  suitable approximation of the Hessian at the step 0.
xk=x0(:); alpha0=1;
if meth==1, hess=varargin{1};
elseif meth==2, hess=varargin{1};
else  hess=[]; end
if ~isempty(h), [nh,mh]=size(h(xk)); end
if ~isempty(g), [ng,mg]=size(g(xk)); else, ng=[]; end
err=tol+1; k=0;
alphak=alpha0; alphak2=alphak/2; told=.1;
while err>tol && k< kmax
P=@(x)Pf(x,fun,g,h,alphak2,ng);
grad_P=@(x)grad_Pf(x,grad_fun,h,g,...
                   grad_h,grad_g,alphak,ng);
if meth==0
options=optimset('TolX',told);
[x,err,kd]=fminsearch(P,xk,options);
err=norm(x-xk);
else
[x,err,kd]=descent(P,grad_P,xk,told,kmaxd,meth,hess);
err=norm(grad_P(x));
end
if kd<kmaxd, alphak=alphak*10; alphak2=alphak/2;
else alphak=alphak*1.5; alphak2=alphak/2; end
k=k+1; xk=x; told=max([tol,told/10]);
end
```

```
end % end of the function penalty
function y=Pf(x,fun,g,h,alphak2,ng)
y=fun(x);
if ~isempty(h), y=y+alphak2*sum((h(x)).^2); end
if ~isempty(g), G=g(x);
for j=1:ng, y=y+alphak2*max([-G(j),0])^2; end
end
end % end of function Pf
function y=grad_Pf(x,grad_fun,h,g,...
                  grad_h,grad_g,alphak,ng)
y=grad_fun(x);
if ~isempty(h), y=y+alphak*grad_h(x)*h(x); end
if ~isempty(g), G=g(x); Gg=grad_g(x);
for j=1:ng
if G(j)<0, y=y+alphak*Gg(:,j)*G(j); end
end, end
end % end of function grad_Pf
```

Example 7.15 Let us solve Problem 2 of the Example 7.13 using Program 7.6. Setting $\mathbf{x}^{(0)} = (1.2, 0.2)$ and tolerance $\overline{\varepsilon} = 10^{-5}$, by the following instructions

```
fun=@(x) 100*(x(2)-x(1).^2).^2+(1-x(1)).^2;
grad_fun=@(x) [-400*(x(2)-x(1)^2)*x(1)-2*(1-x(1));
             200*(x(2)-x(1)^2)];
g=@(x)[-34*x(1)-30*x(2)+19; 10*x(1)-5*x(2)+11;
    3*x(1)+22*x(2)+8];
grad_g=@(x)[-34,10,3;-30,-5,22];
x0=[1.2,.2]; tol=1.e-5; kmax=100; kmaxd=100;
meth=2; hess=eye(2);
[x,err,k]=penalty(fun,grad_fun,[],[],g,grad_g,...
    x0,tol,kmax,kmaxd,meth,hess)
```

after 3 iterations we achieve convergence to the point $(0.41183, 0.16660)$ with a residual on the gradient $\|\nabla_{\mathbf{x}}\mathcal{P}_{\alpha_3}(\mathbf{x}^{(3)})\| \simeq 2.6379\cdot 10^{-7}$. For the solution of the unconstrained minimization problem we have used the program 7.3 `descent`, more precisely the BFGS method described in Section 7.5.4. The constraints at the minimizers are equal to $g_1(\mathbf{x}) = 2.0036e-04$, $g_2(\mathbf{x}) = 1.4285e+01$ and $g_3(\mathbf{x}) = 1.2901e+01$. ■

Example 7.16 To solve Problem 7.3 with the penalty method, let Ω be the circle centered at the origin with radius 2. Let us triangulate Ω with a grid featuring 49 nodes (the vertices) and $Ne = 72$ triangles, as shown in Figure 7.2, left. The 24 boundary nodes are kept fixed, whereas the coordinates of the 25 internal nodes are collected in a vector $\mathbf{x}$ and represent the problem independent variables. The cost function is

$$f(\mathbf{x}) = \sum_{k=1}^{Ne} \frac{1}{\mu_k(\mathbf{x})} = \sum_{k=1}^{Ne} \frac{\sqrt{3}\|\mathrm{A}_k(\mathbf{x})\mathrm{W}^{-1}\|_F^2}{4\det(\mathrm{A}_k(\mathbf{x}))},$$

where we have used definition (7.6), while the inequality constraints are

$$g_k(\mathbf{x}) = \det(\mathrm{A}_k(\mathbf{x})) - \tau \geq 0, \quad k = 1, \dots, Ne,$$

with $\tau = 0.10876$ being twice the value of the area of the smallest triangle of the initial grid. The result shown in Figure 7.2, right, has been obtained after

21 iterations of the penalty algorithm, having set $\bar{\epsilon} = 10^{-8}$ for the stopping test. The maximum number of iterations for the Nelder and Mead method has been fixed to 100. ■

Example 7.17 We solve Problem 7.5 by considering the road network of Figure 7.3. There are $11(= n)$ streets s_j and $7(= p)$ cross roads. We assume that at every minute $M = 20$ cars enter and leave the network, that the length of the streets s_j are collected in the vector $\mathbf{L} = (1, 1, 1.5, 1.5, 1.5, 2.2, 1.5, 1.5, 2.2, 1.5, 2.2)$ km (the jth component refers to the length of s_j street, see Figure 7.3), that the maximum speed allowed on every street is 1 km/min and that the maximum car densities on every street are (the ordered components of the vector) $\boldsymbol{\rho}_m = (60, 40, 20, 60, 60, 40, 60, 20, 40, 20, 60)$. Since we are dealing with a constrained minimization problem with both equality and inequality constraints, we can use the penalty method. The associated unconstrained minimization problem will be solved by the descent method with quasi-Newton directions, for which we need to provide the expression of the gradient of the cost function as well as the constraints. By expressing the cost function f and the functions associated with the constraints in terms of the independent variables ρ_j, we have

$$
\begin{aligned}
&f(\boldsymbol{\rho}) = \left(\sum_{j=1}^{11} \frac{L_j}{v_{j,m}} \frac{\rho_j}{1-\rho_j/\rho_{j,m}}\right) / \sum_{j=1}^{11} \rho_j, \\
&h_1(\boldsymbol{\rho}) = M - \sum_{j=1}^{2} v_{j,m}(1-\rho_j/\rho_{j,m}) \\
&h_2(\boldsymbol{\rho}) = v_{1,m}(1-\rho_1/\rho_{1,m}) - \sum_{j=3}^{4} v_{j,m}(1-\rho_j/\rho_{j,m}) \\
&\dots \\
&h_7(\boldsymbol{\rho}) = \sum_{j=9}^{11} v_{j,m}(1-\rho_j/\rho_{j,m}) - M \\
&g_j(\boldsymbol{\rho}) = \rho_j \qquad\qquad\qquad\qquad j = 1, \dots, 11 \\
&g_{11+j}(\boldsymbol{\rho}) = \rho_{j,m} - \rho_j \qquad\qquad j = 1, \dots, 11.
\end{aligned}
\tag{7.82}
$$

For a given vector $\boldsymbol{\rho}$, the gradient $\nabla f(\boldsymbol{\rho})$ and the matrices $[\nabla h_1(\boldsymbol{\rho}), \dots, \nabla h_p(\boldsymbol{\rho})]$ and $[\nabla g_1(\boldsymbol{\rho}), \dots, \nabla g_n(\boldsymbol{\rho}), \nabla g_{n+1}(\boldsymbol{\rho}), \dots, \nabla g_{2n}(\boldsymbol{\rho})]$ can be built up through 3 distinct MATLAB functions `grad_fun.m`, `grad_h.m`, and `grad_g.m`. By calling the Program `penalty.m`, using an initial vector with unitary components and the tolerance $\bar{\varepsilon} = 10^{-5}$ for the stopping test, after 5 iterations the method converges to the vector

```
rho_opt =
   15.0246     12.8942      4.6535      9.0594      5.8847      9.4996
    0.5278     -0.0000     11.5494      6.9723      8.4272.
```

Its components, ordered by rows and represented in Figure 7.16, provide the densities ρ_j of the cars on the streets s_j that minimize the cost function. The minimum average time founded is $f(\boldsymbol{\rho}_{opt}) = 2.0782$ min. ■

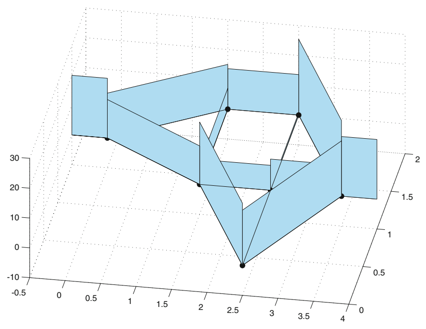


Figure 7.16. The densities ρ_j of the road network Problem 7.5

7.8.2 The augmented Lagrangian method

In this section we address minimization problems with equality constraints only, whence $\mathcal{I}_g = \emptyset$ in (7.69). The function

$$\mathcal{L}_\alpha(\mathbf{x}, \boldsymbol{\lambda}) = f(\mathbf{x}) - \sum_{i \in \mathcal{I}_h} \lambda_i h_i(\mathbf{x}) + \frac{\alpha}{2} \sum_{i \in \mathcal{I}_h} h_i^2(\mathbf{x}) \tag{7.83}$$

obtained from (7.74) is called *augmented Lagrangian*; $\alpha > 0$ is a suitable large coefficient to be assigned.

The augmented Lagrangian method is an iterative method that, at the kth iteration, given α_k and $\boldsymbol{\lambda}^{(k)}$, computes

$$\mathbf{x}^{(k)} = \underset{\mathbf{x} \in \mathbb{R}^n}{\operatorname{argmin}} \, \mathcal{L}_{\alpha_k}(\mathbf{x}, \boldsymbol{\lambda}^{(k)}) \tag{7.84}$$

in such a way that the sequence $\mathbf{x}^{(k)}$ converges to a KKT point (see (7.75)) for the Lagrangian $\mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}) = f(\mathbf{x}) - \sum_{i \in \mathcal{I}_h} \lambda_i h_i(\mathbf{x})$.

The initial values α_0 and $\boldsymbol{\lambda}^{(0)}$ are set arbitrarily. The values for the new iterations are generated as follows. The coefficient α_{k+1} is obtained from α_k proceeding as in the penalty method discussed in Section 7.8.1. On its hand, $\boldsymbol{\lambda}^{(k+1)}$ is computed as follows. We compute $\nabla_{\mathbf{x}} \mathcal{L}_{\alpha_k}(\mathbf{x}, \boldsymbol{\lambda}^{(k)})$ and set it to zero, yielding:

$$\nabla_{\mathbf{x}} \mathcal{L}_{\alpha_k}(\mathbf{x}^{(k)}, \boldsymbol{\lambda}^{(k)}) = \nabla f(\mathbf{x}^{(k)}) - \sum_{i \in \mathcal{I}_h} (\lambda_i^{(k)} - \alpha_k h_i(\mathbf{x}^{(k)})) \nabla h_i(\mathbf{x}^{(k)}) = 0.$$

By comparison with the optimality condition (7.75), we identify the new value of $\lambda_i^{(k+1)}$ as

$$\boxed{\lambda_i^{(k+1)} = \lambda_i^{(k)} - \mu_k h_i(\mathbf{x}^{(k)}).} \tag{7.85}$$

We now obtain $\mathbf{x}^{(k+1)}$ by solving (7.84) with k replaced by $k+1$.

We summarize the algorithm as follows: given α_0 (tipically, $\alpha_0 = 1$), ε_0 (tipically, $\varepsilon_0 = 1/10$), $\overline{\varepsilon} > 0$, $\mathbf{x}_0^{(0)} \in \mathbb{R}^n$ and $\boldsymbol{\lambda}_0^{(0)} \in \mathbb{R}^p$ for $k = 0, 1, \ldots$ until convergence

```
compute an approximation $\mathbf{x}^{(k)}$ to (7.84) using an initial
data $\mathbf{x}_0^{(k)}$ and a tolerance $\varepsilon_k$ on the stopping criterium;
if $\|\nabla_{\mathbf{x}}\mathcal{L}_{\alpha_k}(\mathbf{x}^{(k)}, \boldsymbol{\lambda}^{(k)})\| \leq \overline{\varepsilon}$
    set $\mathbf{x}^* = \mathbf{x}^{(k)}$(convergence achieved)
else
    compute $\lambda_i^{(k+1)}$ by (7.85)
    choose $\alpha_{k+1}$ s.t. $\alpha_{k+1} > \alpha_k$
    choose $\varepsilon_{k+1}$ s.t. $\varepsilon_{k+1} < \varepsilon_k$
    set $\mathbf{x}_0^{(k+1)} = \mathbf{x}^{(k)}$
endif
```
(7.86)

This algorithm is implemented in Program 7.7. Apart from `lambda0` that contains the initial vector $\boldsymbol{\lambda}^{(0)}$ of the Lagrange multipliers, all the other input and output parameters coincide with those of Program 7.6.

Program 7.7. auglagrange: augmented Lagrangian method

```
function [x,err,k]=auglagrange(fun,grad_fun,h,grad_h,...
    x0,lambda0,tol,kmax,kmaxd,meth,varargin)
% AUGLAGRANGE Constrained optimization
%  [X,ERR,K]=AUGLAGRANGE(FUN,GRAD_FUN,H,GRAD_H,...
%  X0,LAMBDA0,TOL,KMAX,KMAXD,METH)
%  computes a local minimizer of the cost function
%  FUN under the constraints H=0, by the augmented
%  Lagrangian method. X0 is the initial point, TOL
%  the tolerance for the stopping test, KMAX the
%  maximum number of allowed iterations.
%  GRAD_FUN and GRAD_H contain the gradient of FUN
%  and H respectively. The solution of the associated
%  unconstrained minimization problem is performed
%  by calling either the Matlab FMINSEARCH function
%  (if METH=0) or the DESCENT function (if METH>0).
%  When METH>0, KMAXD and METH contain respectively
%  the maximum number of allowed iterations for the
%  function DESCENT and the choice of the descent
%  directions. When METH>1
%  [X,ERR,K]=AUGLAGRANGE(FUN,GRAD_FUN,H,GRAD_H,...
%  X0,LAMBDA0,TOL,KMAX,KMAXD,METHi, HESS)
%  is the correct calling instruction.
%  If METH=1 HESS is the function handle associated
%  with the Hessian is required, if METH=2 HESS is a
%  suitable approximation of the Hessian at the step 0.
alpha0=1;
if meth==1, hess=varargin{1};
elseif meth==2, hess=varargin{1};
else, hess=[]; end
err=tol+1; k=0; xk=x0(:); lambdak=lambda0(:);
if ~isempty(h), [nh,mh]=size(h(xk)); end
alphak=alpha0; alphak2=alphak/2; told=0.1;
while err>tol && k< kmax
L=@(x)Lf(x,fun,lambdak,alphak2,h);
grad_L=@(x)grad_Lf(x,grad_fun,lambdak,alphak,h,grad_h);
if meth==0
options=optimset('TolX',told);
[x,err,kd]=fminsearch(L,xk,options);
err=norm(x-xk);
else
[x,err,kd]=descent(L,grad_L,xk,told,kmaxd,meth,hess);
err=norm(grad_L(x));
end
lambdak=lambdak-alphak*h(x);
if kd<kmaxd, alphak=alphak*10; alphak2=alphak/2;
else alphak=alphak*1.5; alphak2=alphak/2; end
k=k+1; xk=x; told=max([tol,told/10]);
end
end % end auglagrange
function y=Lf(x,fun,lambdak,alphak2,h)
y=fun(x);
if ~isempty(h)
y=y-sum(lambdak'*h(x))+alphak2*sum((h(x)).^2); end
end % end function Lf
function y=grad_Lf(x,grad_fun,lambdak,alphak,h,grad_h)
y=grad_fun(x);
if ~isempty(h)
   y=y+grad_h(x)*(alphak*h(x)-lambdak); end
end % end function grad_Lf
```

Example 7.18 To solve Problem 1 of Example 7.13 we use the augmented Lagrangian method by calling Program 7.7 as follows:

```
fun=@(x)0.6*x(1).^2+0.5*x(2).*x(1)-x(2)+3*x(1);
grad_fun=@(x) [1.2*x(1)+0.5*x(2)+3; 0.5*x(1)-1];
h=@(x)x(1).^2+x(2).^2-1;
grad_h=@(x)[2*x(1); 2*x(2)];
x0=[1.2,.2]; tol=1.e-5; kmax=500; kmaxd=100;
p=1; % number of equality constraints
lambda0=rand(p,1); meth=2; hess=eye(2);
[xmin,err,k]=auglagrange(fun,grad_fun,h,grad_h,...
     x0,lambda0,tol,kmax,kmax,meth,hess)
```

We have set the tolerance equal to 10^{-5} for the stopping test, and solved the associated unconstrained minimization problem by quasi-Newton descent directions (therefore setting `meth=2` and `hess=eye(2)`).
After 5 iterations we reach convergence to the point

```
xmin =
    -8.454667252699469e-01
     5.340281045624525e-01
```

The constraint function h at this point is equal to `resh=5.6046-10`. The solution to this problem is reported in Figure 7.15, left.

Should we use the penalty method instead, leaving unchanged all the other settings, we would obtain convergence after 6 iterations to the point

```
xmin =
    -8.454715822058602e-01
     5.340328869427682e-01
```

with the value of h therein equal to `resh=1.3320e-04`. The latter value is larger by 6 orders of magnitude than the one obtained using the augmented Lagrangian method. Since this behaviour occurs quite often, the augmented Lagrangian method is in general preferable in case of minimization problems featuring only equality constraints. ■

See Exercises 7.12-7.14.

Let us summarize

1. For a constrained minimization problem, the minimizers should be sought for among the KKT points associated with the Lagrangian function, or among the points where the LICQ condition fails to be satisfied;
2. a quadratic programming problem is one for which the cost function is quadratic and the constraints are linear. Under suitable assumptions on the matrix associated with the quadratic terms and on the constraint functions, it admits a unique minimizer that can be obtained by solving a linear system;
3. a constrained minimization problem can be turned into an unconstrained one using a suitable penalty function. The corresponding penalized problem can be severely ill-conditioned because of the large value that is tipically assigned to the penalty parameter;
4. the augmented Lagrangian method is a penalty method suitable for the search of KKT points.

7.9 What we haven't told you

Large scale optimization problems are especially demanding in terms of computational time and storage requirements. Both line search and trust region methods require the factorization of the Hessian matrix or the construction of suitable approximations that might be dense even when the Hessian is sparse. Special variants featuring limited memory of the methods illustrated above have been developed, based on Conjugate Gradient and Lanczos iterations. See for instance [Ste83, NW06, GOT05].

A classical and efficient method for the solution of constrained minimization problems is the *Sequential Quadratic Programming* (SQP), which transforms a minimization problem with cost function f and arbitrary constraints into the successive solution of quadratic programming problems. At every iteration, f is approximated by a quadratic function like (7.76), then one looks for the KKT points of the associated Lagrangian function (see for instance [Fle10], [NW06]).

In case of inequality constraints solely, the *barrier methods* represent an alternative to penalty methods: the cost function is modified by adding a function depending on the inequality constraints which inhibits an admissible point $\mathbf{x} \in \Omega$ to generate a successive point which is not admissible. This barrier function is defined only at the interior of the admissible set and is unbounded on the boundary of Ω. These methods require the initial point to be admissible, a condition hard to be fulfilled. For a more in depth presentation we refer to [Ter10].

7.10 Exercises

Exercise 7.1 Compute the minimum of $f(x) = (x-1)e^{-x^2}$ using the golden section method with or without quadratic interpolation.

Exercise 7.2 Two ships leave the harbour at the same time and move along trajectories respectively described by the parametric curves

$$\gamma_1(t) = \begin{cases} 7\cos\left(\frac{t}{3}+\frac{\pi}{2}\right)+5 \\ -4\sin\left(\frac{t}{3}+\frac{\pi}{2}\right)-3 \end{cases}, \qquad \gamma_2(t) = \begin{cases} 6\cos\left(\frac{t}{6}-\frac{\pi}{3}\right)-4 \\ -6\sin\left(\frac{t}{3}-\frac{\pi}{3}\right)+5 \end{cases}.$$

The parameter $t > 0$ represents the time (in hours), whereas the positions are expressed in miles with respect to the origin of the reference framework. Find the minimum distance between the two ships along all their motion.

Exercise 7.3 Compute the global minima of $f(\mathbf{x}) = x_1^4 + x_2^4 + x_1^3 + 3x_1x_2^2 - 3x_1^2 - 3x_2^2 + 10$ using the Nelder and Mead method.

Exercise 7.4 By setting $x^{(0)} = 3/2$, $d^{(k)} = (-1)^{k+1}$, and $\alpha_k = 2 + 2/3^{k+1}$, show that the descent method generates a sequence that does not converge to the minimizer of $f(x) = x^4$ even though $\{f(x^{(k)})\}$ is monotonically decreasing. Show moreover that the steplengths α_k do not fulfill the Wolfe conditions (7.43).

Exercise 7.5 Show that the same conclusions drawn for the previous exercise hold by taking $x^{(0)} = -2$, $d^{(k)} = 1$, and $\alpha_k = 3^{-(k+1)}$.

Exercise 7.6 Approximate the minimizer of the Rosenbrock function defined in Example 7.3 using the descent method with different choices of the descent directions (7.35)–(7.38). Set $\mathbf{x}^{(0)} = (-1.2, 1)$ and $\varepsilon = 10^{-8}$ as tolerance for the stopping test, plot the convergence histories for the different choices of the descent directions and comment on the efficiency of the different methods.

Exercise 7.7 Compute the minimum of $f(\mathbf{x}) = (x_1^2 - x_1^3 x_2 - 2x_2 + 2x_1 x_2^2)^2 + (3 - x_1 x_2)^2$ using the BFGS method and the trust region method with quasi-Newton directions to solve problem (7.54). As initial guess try $\mathbf{x}^{(0)} = (2, -1)$, or $\mathbf{x}^{(0)} = (2, 1)$, or else $\mathbf{x}^{(0)} = (-1, -1)$.

Exercise 7.8 Show that the Gauss-Newton method (7.63) can be reformulated as follows: for $k = 0, 1, \dots$ until convergence, solve

$$\min_{\mathbf{x}\in\mathbb{R}^n} \frac{1}{2}\|\widetilde{\mathbf{R}}_k(\mathbf{x})\|^2 \text{ with } \widetilde{\mathbf{R}}_k(\mathbf{x}) \text{ defined in (7.64).} \tag{7.87}$$

Exercise 7.9 Consider the Gauss-Newton method of Section 7.7.1. Show that if $\mathrm{J}_{\mathbf{R}}(\mathbf{x}^{(k)})$ has full rank, then the solution $\boldsymbol{\delta}\mathbf{x}^{(k)}$ of $(7.63)_1$ is a descent direction for the function f defined in (7.61).

Exercise 7.10 Consider the table

t_i	0.055	0.181	0.245	0.342	0.419	0.465	0.593	0.752
y_i	2.80	1.76	1.61	1.21	1.25	1.13	0.52	0.28

and find the least squares approximation $\phi(t) = x_1 + x_2 t + x_3 t^2 + x_4 e^{-x_5 t}$ (with unknown coefficients $x_1,\ x_2, \dots, x_5$) of the data set (t_i, y_i).

Exercise 7.11 Prove that the function $\tilde{\Phi}_k(\mathbf{x})$ defined in (7.65) is a quadratic approximation of Φ obtained by approximating $\mathbf{R}(\mathbf{x})$ with its linear model (7.64).

Exercise 7.12 We look for the optimal positioning of the warehouse that has to provide goods to three selling points whose coordinates are reported in the table below:

Selling point	coordinates (x_i, y_i) (km)	annual deliveries (units)
1	(6,3)	140
2	(-9,9)	134
3	(-8,-5)	88

The warehouse must be allocated within the region $\Omega = \{(x, y) \in \mathbb{R}^2 :\ y \leq x - 10\}$.

Exercise 7.13 Compute the minimum of the Quadratic Programming problem (7.76) featuring only equality constraints, with

$$\mathrm{A} = \begin{bmatrix} 2 & -1 & 1 \\ -1 & 3 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 1 \\ -2 \\ -1 \end{bmatrix}, \qquad \mathrm{C} = \begin{bmatrix} 2 & -2 & 0 \\ 2 & 1 & -3 \end{bmatrix}, \quad \mathbf{d} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}.$$

Exercise 7.14 A material point moves with speed $v(x, y) = (\sin(\pi xy) + 1)(2x + 3y + 4)$ along an elliptic trajectory whose equation is $x^2/4 + y^2 = 1$. Find the maximum value of the velocity reached by the point as well as the corresponding position.

8

Ordinary differential equations

A differential equation is an equation involving one or more derivatives of an unknown function. If all derivatives are taken with respect to a single independent variable we call it an *ordinary differential equation*, whereas we have a *partial differential equation* when partial derivatives are present.

A differential equation (ordinary or partial) has *order p* if p is the maximum order of differentiation that is present. The next chapter will be devoted to the study of partial differential equations, whereas in the present chapter we will deal with ordinary differential equations of first order.

8.1 Some representative problems

Ordinary differential equations describe the evolution of many phenomena in various fields, as we can see from the following four examples.

Problem 8.1 (Thermodynamics) Consider a body having internal temperature T which is set in an environment with constant temperature T_e. Assume that its mass m is concentrated in a single point. Then the heat transfer between the body and the external environment can be described by the Stefan-Boltzmann law

$$v(t) = \epsilon\gamma S(T^4(t) - T_e^4),$$

where t is the time variable, ϵ the Stefan-Boltzmann constant (equal to $5.6\cdot 10^{-8}\mathrm{J}/(\mathrm{m}^2\mathrm{K}^4\mathrm{s})$ where J stands for Joule, K for Kelvin and, obviously, m for meter, s for second), γ is the emissivity constant of the body, S the area of its surface and v is the rate of the heat transfer. The rate of variation of the energy $E(t) = mCT(t)$ (where C denotes the specific heat of the material constituting the body) equals, in absolute value, the rate v. Consequently, setting $T(0) = T_0$, the computation of $T(t)$ requires the solution of the ordinary differential equation

$$\frac{dT}{dt} = -\frac{v}{mC}. \tag{8.1}$$

See Exercise 8.15 for its solution. ■

Problem 8.2 (Population dynamics) Consider a population of bacteria in a confined environment in which no more than B elements can coexist. Assume that, at the initial time, the number of individuals is equal to $y_0 \ll B$ and the growth rate of the bacteria is a positive constant C. In this case the rate of change of the population is proportional to the number of existing bacteria, under the restriction that the total number cannot exceed B. This is expressed by the differential equation

$$\frac{dy}{dt} = Cy\left(1 - \frac{y}{B}\right), \tag{8.2}$$

whose solution $y = y(t)$ denotes the number of bacteria at time t.

Assuming that two populations y_1 and y_2 be in competition, instead of (8.2) we would have

$$\begin{aligned}\frac{dy_1}{dt} &= C_1 y_1 (1 - b_1 y_1 - d_2 y_2), \\ \frac{dy_2}{dt} &= -C_2 y_2 (1 - b_2 y_2 - d_1 y_1),\end{aligned} \tag{8.3}$$

where C_1 and C_2 represent the growth rates of the two populations. The coefficients d_1 and d_2 govern the type of interaction between the two populations, while b_1 and b_2 are related to the available quantity of nutrients. The above equations (8.3) are called the Lotka-Volterra equations and form the basis of various applications. For their numerical solution, see Example 8.7. ■

Problem 8.3 (Baseball trajectory) We want to simulate the trajectory of a ball from the pitcher to the catcher. By adopting the reference frame of Figure 8.1, the equations describing the ball motion are (see [Ada90], [GN06])

$$\frac{d\mathbf{x}}{dt} = \mathbf{v}, \qquad \frac{d\mathbf{v}}{dt} = \mathbf{F},$$

where $\mathbf{x}(t) = (x(t), y(t), z(t))^T$ designates the position of the ball at time t, $\mathbf{v}(t) = (v_x(t), v_y(t), v_z(t))^T$ its velocity, while $\mathbf{F}$ is the vector whose components are

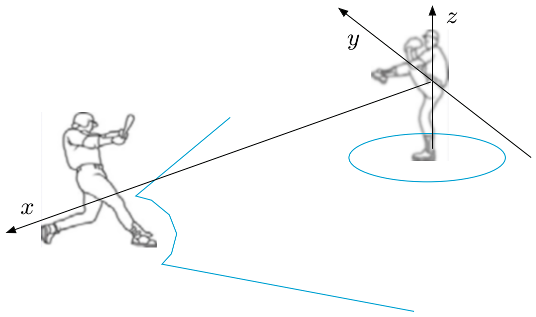


Figure 8.1. The reference frame adopted for Problem 8.3

$$
\begin{aligned}
F_x &= -F(v)vv_x + B\omega(v_z \sin\phi - v_y \cos\phi),\\
F_y &= -F(v)vv_y + B\omega v_x \cos\phi, \qquad (8.4)\\
F_z &= -g - F(v)vv_z - B\omega v_x \sin\phi.
\end{aligned}
$$

v is the modulus of $\mathbf{v}$, $B = 4.1\ 10^{-4}$ a normalized constant, ϕ is the pitching angle, ω is the modulus of the angular velocity impressed to the ball from the pitcher. $F(v)$ is a friction coefficient, normally defined as ([GN06])

$$F(v) = 0.0039 + \frac{0.0058}{1 + e^{(v-35)/5}}.$$

The solution of this system of ordinary differential equations is postponed to Exercise 8.20. ■

Problem 8.4 (Electrical circuits) Consider the electrical circuit of Figure 8.2. We want to compute the function $v(t)$ representing the potential drop at the ends of the capacitor C starting from the initial time $t = 0$ at which the switch I has been turned off. Assume that the inductance L can be expressed as an explicit function of the current intensity i, that is $L = L(i)$. The Ohm law yields

$$e - \frac{d(i_1 L(i_1))}{dt} = i_1 R_1 + v,$$

where R_1 is a resistance. By assuming the current fluxes to be directed as indicated in Figure 8.2, upon differentiating with respect to t both sides of the Kirchoff law $i_1 = i_2 + i_3$ and noticing that $i_3 = C dv/dt$ and $i_2 = v/R_2$, we find the further equation

$$\frac{di_1}{dt} = C\frac{d^2v}{dt^2} + \frac{1}{R_2}\frac{dv}{dt}.$$

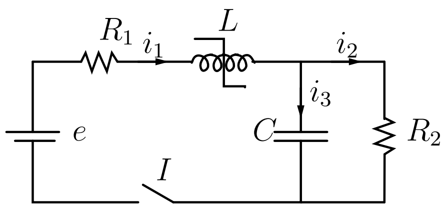


Figure 8.2. The electrical circuit of Problem 8.4

We have therefore found a system of two differential equations whose solution allows the description of the time variation of the two unknowns i_1 and v. The second equation has order two. For its solution see Example 8.8. ■

8.2 The Cauchy problem

We confine ourselves to first order differential equations, as an equation of order $p > 1$ can always be reduced to a system of p equations of order 1. The case of first order systems will be addressed in Section 8.9.

An ordinary differential equation in general admits an infinite number of solutions. In order to fix one of them we must impose a further condition which prescribes the value taken by this solution at a given point of the integration interval. For instance, the equation (8.2) admits the family of solutions $y(t) = B\psi(t)/(1 + \psi(t))$ with $\psi(t) = e^{Ct+K}$, K being an arbitrary constant. If we impose the condition $y(0) = 1$, we pick up the unique solution corresponding to the value $K = \ln[1/(B - 1)]$.

We will therefore consider the solution of the so-called *Cauchy problem* which takes the following form:

find $y : I \subset \mathbb{R} \to \mathbb{R}$ such that

$$\begin{cases} y'(t) = f(t, y(t)) & \forall t \in I, \\ y(t_0) = y_0, \end{cases} \tag{8.5}$$

where I is an interval, $f : I \times \mathbb{R} \to \mathbb{R}$ is a given function, y' denotes the derivative of y with respect to t, t_0 is a point of I and y_0 a given value which is called the *initial data*.

In the following proposition we report a classical result of Analysis.

Proposition 8.1 *Assume that the function $f(t,y)$ is*

1. *continuous with respect to both its arguments;*
2. *Lipschitz-continuous with respect to its second argument, that is, there exists a positive constant L (named Lipschitz constant) such that*

$$|f(t,y_1) - f(t,y_2)| \leq L|y_1 - y_2| \quad \forall t \in I,\ \forall y_1, y_2 \in \mathbb{R}.$$

Then the solution $y = y(t)$ of the Cauchy problem (8.5) exists, is unique and belongs to $C^1(I)$.

Unfortunately, explicit solutions are available only for very special types of ordinary differential equations. In some other cases, the solution is available only in implicit form. This is, for instance, the case with the equation $y' = (y-t)/(y+t)$ whose solution satisfies the implicit relation

$$\frac{1}{2}\ln(t^2+y^2) + \mathrm{arctg}\frac{y}{t} = C,$$

where C is an arbitrary constant. In some other circumstances the solution is not even representable in implicit form, as in the case of the equation $y' = e^{-t^2}$ whose general solution can only be expressed through a series expansion. For all these reasons, we seek numerical methods capable of approximating the solution of *every* family of ordinary differential equations for which solutions do exist.

The common strategy of all these methods consists of subdividing the integration interval $I = [t_0, T]$, with $T < +\infty$, into N_h intervals of length $h = (T - t_0)/N_h$; h is called the *discretization step*, or *time-step*, or *steplength*. Then, at each *node* $t_n = t_0 + nh$ (for $n = 1, \ldots, N_h$) we seek the unknown value u_n which approximates $y_n = y(t_n)$. The set of values $\{u_0 = y_0, u_1, \ldots, u_{N_h}\}$ represents our *numerical solution*.

8.3 Euler methods

A classical method, the *forward Euler* method, generates the numerical solution as follows

$$\boxed{u_{n+1} = u_n + hf_n, \qquad n = 0, \ldots, N_h - 1} \tag{8.6}$$

where we have used the shorthand notation $f_n = f(t_n, u_n)$. This method is obtained by considering the differential equation (8.5) at every node t_n, $n = 1, \ldots, N_h$ and replacing the exact derivative $y'(t_n)$ by means of the incremental ratio (4.4).

In a similar way, using this time the incremental ratio (4.8) to approximate $y'(t_{n+1})$, we obtain the *backward Euler* method

$$\boxed{u_{n+1} = u_n + hf_{n+1}, \qquad n = 0, \ldots, N_h - 1} \tag{8.7}$$

Both methods provide an instance of a *one-step method* since for computing the numerical solution u_{n+1} at the node t_{n+1} we only need the information related to the previous node t_n. More precisely, in the forward Euler method u_{n+1} depends exclusively on the value u_n previously computed, whereas in the backward Euler method it depends also on itself through the value f_{n+1}. For this reason the first method is called the *explicit* Euler method and the second one the *implicit* Euler method.

For instance, the discretization of (8.2) by the forward Euler method requires at every step the simple computation of

$$u_{n+1} = u_n + hCu_n \left(1 - u_n/B\right),$$

whereas using the backward Euler method we must solve the nonlinear equation

$$u_{n+1} = u_n + hCu_{n+1} \left(1 - u_{n+1}/B\right).$$

Thus, implicit methods are more costly than explicit methods, since, if the function f in (8.5) is not linear, at every time level t_{n+1} we must solve a nonlinear problem to compute u_{n+1}. However, we will see that implicit methods enjoy better stability properties than explicit ones.

The forward Euler method is implemented in Program 8.1; the integration interval is `tspan = [t0,tfinal]`, `odefun` is the function handle associated with the function $f(t, y(t))$ which depends on the variables t and y.

Program 8.1. feuler: forward Euler method

```
function [t,u]=feuler(odefun,tspan,y0,Nh,varargin)
%FEULER Solves differential equations using the forward
%   Euler method.
%   [T,Y]=FEULER(ODEFUN,TSPAN,Y0,NH) with TSPAN=[T0,TF]
%   integrates the system of differential equations
%   y'=f(t,y) from time T0 to TF with initial condition
%   Y0 using the forward Euler method on an equispaced
%   grid of NH intervals.
%   Function ODEFUN(T,Y) must return a vector, whose
%   elements hold the evaluation of f(t,y), of the
%   same dimension of Y.
%   Each row in the solution array Y corresponds to a
%   time returned in the  column vector T.
%   [T,Y] = FEULER(ODEFUN,TSPAN,Y0,NH,P1,P2,...) passes
%   the additional parameters P1,P2,... to the function
%   ODEFUN as ODEFUN(T,Y,P1,P2...).
```

```
h=(tspan(2)-tspan(1))/Nh;
y=y0(:); % always creates a column vector
w=y; u=y.';
tt=linspace(tspan(1),tspan(2),Nh+1);
for t = tt(1:end-1)
 w=w+h*odefun(t,w,varargin{:});
 u = [u; w.'];
end
t=tt';
return
```

The backward Euler method is implemented in Program 8.2. Note that we have used the function `fsolve` for the solution of the nonlinear problem at each step. As initial data for `fsolve` we use the last computed value of the numerical solution.

Program 8.2. beuler: backward Euler method

```
function [t,u]=beuler(odefun,tspan,y0,Nh,varargin)
%BEULER Solves differential equations using the
%  backward Euler method.
%  [T,Y]=BEULER(ODEFUN,TSPAN,Y0,NH) with TSPAN=[T0,TF]
%  integrates the system of differential equations
%  y'=f(t,y) from time T0 to TF with initial condition
%  Y0 using the backward Euler method on an equispaced
%  grid of NH intervals.
%  Function ODEFUN(T,Y) must return a vector, whose
%  elements hold the evaluation of f(t,y), of the
%  same dimension of Y.
%  Each row in the solution array Y corresponds to a
%  time returned in the  column vector T.
%  [T,Y] = BEULER(ODEFUN,TSPAN,Y0,NH,P1,P2,...) passes
%  the additional parameters P1,P2,... to the function
%  ODEFUN as ODEFUN(T,Y,P1,P2...).
tt=linspace(tspan(1),tspan(2),Nh+1);
y=y0(:); % always create a vector column
u=y.';
global glob_h glob_t glob_y glob_odefun;
glob_h=(tspan(2)-tspan(1))/Nh;
glob_y=y;
glob_odefun=odefun;
glob_t=tt(2);

if ( exist('OCTAVE_VERSION') )
o_ver=OCTAVE_VERSION;
version=str2num([o_ver(1),o_ver(3),o_ver(5)]);
end

if ( ~exist('OCTAVE_VERSION') | version >= 320 )
options=optimset;
options.Display='off';
options.TolFun=1.e-12;
options.MaxFunEvals=10000;
end
for glob_t=tt(2:end)
if ( exist('OCTAVE_VERSION') & version < 320 )
  w = fsolve('beulerfun',glob_y);
```

```
else
  w = fsolve(@(w) beulerfun(w),glob_y,options);
end
  u = [u; w.'];
  glob_y = w;
end
t=tt';
clear glob_h glob_t glob_y glob_odefun;
end

function [z]=beulerfun(w)
  global glob_h glob_t glob_y glob_odefun;
  z=w-glob_y-glob_h*glob_odefun(glob_t,w);
end
```

8.3.1 Convergence analysis

A numerical method is *convergent* if

$$\boxed{\forall n = 0, \ldots, N_h, \qquad |y_n - u_n| \leq C(h)} \tag{8.8}$$

where $C(h)$ is infinitesimal with respect to h when h tends to zero. If $C(h) = \mathcal{O}(h^p)$ for some $p > 0$ (that is there exists a positive constant c such that $C(h) \leq ch^p$ and p is the maximum integer for which this inequality holds), then we say that the method converges with *order* p.

In order to verify that the forward Euler method converges, we write the error as follows:

$$e_n = y_n - u_n = (y_n - u_n^*) + (u_n^* - u_n), \tag{8.9}$$

where

$$u_n^* = y_{n-1} + hf(t_{n-1}, y_{n-1})$$

denotes the numerical solution at time t_n which we would obtain starting from the exact solution at time t_{n-1}; see Figure 8.3. The term $y_n - u_n^*$ in (8.9) represents the error produced by a single step of the forward Euler method (this error is infinitesimal thanks to the consistency property), whereas the term $u_n^* - u_n$ represents the propagation from t_{n-1} to t_n of the error accumulated at the previous time level t_{n-1} (this propagation is bounded thanks to the stability property). The method converges provided both terms tend to zero as $h \to 0$; otherwise said, convergence is assured if the method is both consistent and stable.

Assuming that the second order derivative of y exists and is continuous, thanks to (4.6) we find that there exists $\xi_n \in (t_{n-1}, t_n)$ such that

$$y_n - u_n^* = \frac{h^2}{2} y''(\xi_n). \tag{8.10}$$

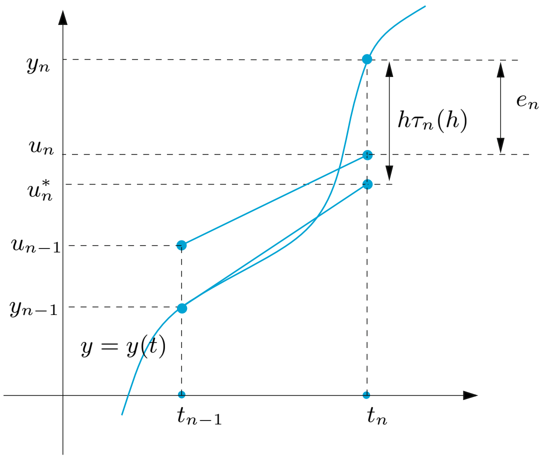


Figure 8.3. Geometrical representation of a step of the forward Euler method

The quantity

$$\tau_n(h) = (y_n - u_n^*)/h$$

is named local truncation error of the forward Euler method.

More in general, the *local truncation error* of a given method represents (up to a factor $1/h$) the error that would be generated by forcing the exact solution to satisfy that specific numerical scheme.

The *global truncation error* (or, more simply, *truncation error*) is defined as

$$\tau(h) = \max_{n=0,\ldots,N_h} |\tau_n(h)|.$$

In view of (8.10), the truncation error for the forward Euler method takes the following form

$$\tau(h) = Mh/2, \tag{8.11}$$

where $M = \max_{t\in[t_0,T]} |y''(t)|$.

From (8.10) we deduce that $\lim_{h\to 0} \tau(h) = 0$, and a method for which this happens is said to be *consistent*. Further, we say that it is consistent with order p if $\tau(h) = \mathcal{O}(h^p)$ for a suitable integer $p \geq 1$.

Consider now the other term in (8.9). We have

$$u_n^* - u_n = e_{n-1} + h\left[f(t_{n-1}, y_{n-1}) - f(t_{n-1}, u_{n-1})\right]. \tag{8.12}$$

Since f is Lipschitz-continuous with respect to its second argument, we obtain

$$|u_n^* - u_n| \leq (1 + hL)|e_{n-1}|.$$

If $e_0 = 0$, the previous relations yield

$$\begin{aligned}|e_n| &\leq |y_n - u_n^*| + |u_n^* - u_n| \\ &\leq h|\tau_n(h)| + (1 + hL)|e_{n-1}| \\ &\leq \left[1 + (1 + hL) + \ldots + (1 + hL)^{n-1}\right] h\tau(h) \\ &= \frac{(1 + hL)^n - 1}{L}\tau(h) \leq \frac{e^{L(t_n - t_0)} - 1}{L}\tau(h).\end{aligned}$$

We have used the identity

$$\sum_{k=0}^{n-1}(1 + hL)^k = [(1 + hL)^n - 1]/hL,$$

the inequality $1 + hL \leq e^{hL}$ and we have observed that $nh = t_n - t_0$. Therefore we find

$$|e_n| \leq \frac{e^{L(t_n - t_0)} - 1}{L}\frac{M}{2}h \qquad \forall n = 0, \ldots, N_h, \tag{8.13}$$

and thus we can conclude that *the forward Euler method converges with order 1*. We can note that the order of this method coincides with the order of its local truncation error. This property is shared by many numerical methods for the numerical solution of ordinary differential equations. The convergence estimate (8.13) is now obtained by simply requiring f to be Lipschitz-continuous.

A better estimate, precisely

$$|e_n| \leq Mh(t_n - t_0)/2, \tag{8.14}$$

holds if $\partial f/\partial y$ exists and satisfies the further requirement $\partial f(t, y)/\partial y \leq 0$ for all $t \in [t_0, T]$ and all $-\infty < y < \infty$. Indeed, in that case, using Taylor expansion, from (8.12) we obtain

$$u_n^* - u_n = \left(1 + h\frac{\partial f}{\partial y}(t_{n-1}, \eta_n)\right) e_{n-1},$$

where η_n belongs to the interval whose endpoints are y_{n-1} and u_{n-1}, thus $|u_n^* - u_n| \leq |e_{n-1}|$, provided the inequality

$$\boxed{0 < h < 2/\max_{t\in[t_0,T]}\left|\frac{\partial f}{\partial y}(t, y(t))\right|} \tag{8.15}$$

holds. Then $|e_n| \leq |y_n - u_n^*| + |e_{n-1}| \leq nh\tau(h) + |e_0|$, whence (8.14) owing to (8.11) and to the fact that $e_0 = 0$. The limitation (8.15) on the step h is in fact a *stability condition*, as we will see in the sequel.

Remark 8.1 (Consistency) The property of consistency is necessary in order to get convergence. Actually, should it be violated, at each step the numerical method would generate an error which is not infinitesimal with respect to h. The accumulation with the previous errors would inhibit the global error to converge to zero when $h \to 0$. ■

For the backward Euler method the local truncation error reads

$$\tau_n(h) = \frac{1}{h}[y_n - y_{n-1} - hf(t_n, y_n)].$$

Still using the Taylor expansion one obtains

$$\tau_n(h) = -\frac{h}{2}y''(\xi_n)$$

for a suitable $\xi_n \in (t_{n-1}, t_n)$, provided $y \in C^2$. Thus also the backward Euler method converges with order 1 with respect to h.

Example 8.1 Consider the Cauchy problem

$$\begin{cases} y'(t) = \cos(2y(t)), & t \in (0,1], \\ y(0) = 0, \end{cases} \tag{8.16}$$

whose solution is $y(t) = \frac{1}{2}\arcsin((e^{4t}-1)/(e^{4t}+1))$. We solve it by the forward Euler method (Program 8.1) and the backward Euler method (Program 8.2). By the following commands we use different values of h, $1/2$, $1/4, 1/8, \ldots, 1/512$:

```
tspan=[0,1]; y0=0; f=@(t,y) cos(2*y);
u=@(t) 0.5*asin((exp(4*t)-1)./(exp(4*t)+1));
Nh=2;
for k=1:10
    [t,ufe]=feuler(f,tspan,y0,Nh);
    fe(k)=abs(ufe(end)-u(t(end)));
    [t,ube]=beuler(f,tspan,y0,Nh);
    be(k)=max(abs(ube-u(t)));
    Nh = 2*Nh;
end
```

The errors computed at the point $t = 1$ are stored in the variable `fe` (forward Euler) and `be` (backward Euler), respectively. Then we apply formula (1.12) to estimate the order of convergence. Using the following commands

```
p=log(abs(fe(1:end-1)./fe(2:end)))/log(2); p(1:2:end)

  1.2898    1.0349    1.0080    1.0019    1.0005


  0.8770    0.9649    0.9908    0.9978    0.9994
```

we can verify that both methods are convergent with order 1. ■

Remark 8.2 (Roundoff errors effects) The error estimate (8.13) was derived by assuming that the numerical solution $\{u_n\}$ is obtained in exact arithmetic. Should we account for the (inevitable) roundoff-errors, the error might blow up like $\mathcal{O}(1/h)$ as h approaches 0 (see, e.g., [Atk89]). This circumstance suggests that it might be unreasonable to go below a certain threshold h^* (which is actually extremely tiny) in practical computations. ■

See the Exercises 8.1-8.3.

8.4 The Crank-Nicolson method

By combining the generic steps of the forward and backward Euler methods we find the so-called *Crank-Nicolson method*

$$\boxed{u_{n+1} = u_n + \frac{h}{2}[f_n + f_{n+1}], \quad n = 0, \ldots, N_h - 1} \tag{8.17}$$

This method can also be derived by applying the fundamental theorem of integration (which we recalled in Section 1.5.3) to the Cauchy problem (8.5), obtaining

$$y_{n+1} = y_n + \int_{t_n}^{t_{n+1}} f(t, y(t))\ dt, \tag{8.18}$$

and then approximating the integral by the trapezoidal rule (4.19).

The local truncation error of the Crank-Nicolson method satisfies

$$\begin{aligned}\tau_n(h) &= \frac{1}{h}[y(t_n) - y(t_{n-1})] - \frac{1}{2}\left[f(t_n, y(t_n)) + f(t_{n-1}, y(t_{n-1}))\right]\\ &= \frac{1}{h}\int_{t_{n-1}}^{t_n} f(t, y(t))\ dt - \frac{1}{2}\left[f(t_n, y(t_n)) + f(t_{n-1}, y(t_{n-1}))\right].\end{aligned}$$

The last equality follows from (8.18) and expresses, up to a factor of $1/h$, the error associated with the trapezoidal rule for numerical integration (4.19). If we assume that $y \in C^3$ and use (4.20), we deduce that

$$\tau_n(h) = -\frac{h^2}{12}y'''(\xi_n) \text{ for a suitable } \xi_n \in (t_{n-1}, t_n). \tag{8.19}$$

Thus the Crank-Nicolson method is consistent with order 2, i.e. its local truncation error tends to 0 as h^2. Using a similar approach to that followed for the forward Euler method, we can show that the Crank-Nicolson method is convergent with order 2 with respect to h.

The Crank-Nicolson method is implemented in the Program 8.3. Input and output parameters are the same as for the Euler methods.

Program 8.3. cranknic: Crank-Nicolson method

```
function [t,u]=cranknic(odefun,tspan,y0,Nh,varargin)
%CRANKNIC  Solves differential equations using the
%  Crank-Nicolson method.
%  [T,Y]=CRANKNIC(ODEFUN,TSPAN,Y0,NH)  with
%  TSPAN=[T0,TF] integrates the system of differential
%  equations y'=f(t,y) from time T0 to TF with initial
%  condition Y0 using the Crank-Nicolson method on an
%  equispaced grid of NH intervals.
%  Function ODEFUN(T,Y) must return a vector, whose
%  elements hold the evaluation of f(t,y), of the
%  same dimension of Y.
%  Each row in the solution array Y corresponds to a
%  time returned in the  column vector T.
%  [T,Y] = CRANKNIC(ODEFUN,TSPAN,Y0,NH,P1,P2,...)
%  passes the additional parameters P1,P2,... to the
%  function ODEFUN as ODEFUN(T,Y,P1,P2...).
tt=linspace(tspan(1),tspan(2),Nh+1);
y=y0(:); % always create a vector column
u=y.';
global glob_h glob_t glob_y glob_odefun;
glob_h=(tspan(2)-tspan(1))/Nh;
glob_y=y;
glob_odefun=odefun;
if ( exist('OCTAVE_VERSION') )
o_ver=OCTAVE_VERSION;
version=str2num([o_ver(1),o_ver(3),o_ver(5)]);
end

if( ~exist('OCTAVE_VERSION')  | version >= 320 )
 options=optimset;
 options.Display='off';
 options.TolFun=1.e-12;
 options.MaxFunEvals=10000;
end
for glob_t=tt(2:end)
if ( exist('OCTAVE_VERSION') & version < 320 )
  w = fsolve('cranknicfun',glob_y);
else
  w = fsolve(@(w) cranknicfun(w),glob_y,options);
end
  u = [u; w.'];
  glob_y = w;
end
t=tt';
clear glob_h glob_t glob_y glob_odefun;
end

function z=cranknicfun(w)
  global glob_h glob_t glob_y glob_odefun;
  z=w - glob_y - ...
    0.5*glob_h*(glob_odefun(glob_t,w) + ...
    glob_odefun(glob_t-glob_h,glob_y));
end
```

Example 8.2 Let us solve the Cauchy problem (8.16) by using the Crank-Nicolson method with the same values of h as used in Example 8.1. The results show that the error tends to zero with order $p = 2$ with respect to h:

```
y0=0;  tspan=[0 1]; N=2; f=@(t,y) cos(2*y);
y=@(t) 0.5*asin((exp(4*t)-1)./(exp(4*t)+1));
for k=1:10
  [tt,u]=cranknic(f,tspan,y0,N);
  e(k)=max(abs(u-y(tt))); N=2*N;
end
p=log(abs(e(1:end-1)./e(2:end)))/log(2); p(1:2:end)

    1.9627    1.9986    2.0001    1.9999    2.0000
```

■

See the Exercises 8.4-8.5.

8.5 Zero-stability

Commonly speaking, by stability of a numerical scheme we mean its capability to keep the effects on the solution of data perturbations under control.

Among several concepts of stability, there is the zero-stability, which guarantees that, in a *fixed bounded interval*, small perturbations of data yield bounded perturbations of the numerical solution *when* $h \to 0$.

More precisely, a numerical method for the approximation of problem (8.5), with $I = [t_0, T]$, is *zero-stable* if

$\exists h_0 > 0$, $\exists C > 0$, $\exists \varepsilon_0 > 0$ s.t. $\forall h \in (0, h_0], \forall \varepsilon \in (0, \varepsilon_0]$, if $|\rho_n| \leq \varepsilon, 0 \leq n \leq N_h$, then

$$|z_n - u_n| \leq C\varepsilon, \qquad 0 \leq n \leq N_h, \tag{8.20}$$

where:
- C is a constant which might depend on the length $T - t_0$ of the integration interval I, but is independent of h;
- z_n is the solution that would be obtained by applying the numerical method at hand to a *perturbed* problem;
- ρ_n denotes the size of the perturbation introduced at the nth step;
- ε indicates the maximum size of the perturbation.

Obviously, ε_0 and ε must be small enough to guarantee that the perturbed problem still has a unique solution on the integration interval I.

For instance, in the case of the forward Euler method u_n satisfies the problem

$$\begin{cases} u_{n+1} = u_n + hf(t_n, u_n), \quad n = 0, \dots, N_h - 1 \\ u_0 = y_0, \end{cases} \tag{8.21}$$

whereas z_n satisfies the perturbed problem

$$\begin{cases} z_{n+1} = z_n + h\left[f(t_n, z_n) + \rho_{n+1}\right], \quad n = 0, \ldots, N_h - 1 \\ z_0 = y_0 + \rho_0. \end{cases} \tag{8.22}$$

For a consistent one-step method zero-stability follows for the property of f to be Lipschitz-continuous with respect to its second argument (see, e.g. [QSS07]). In that case, the constant C that appears in (8.20) depends on $\exp((T - t_0)L)$, where L is the Lipschitz constant.

However, this is not necessarily true for other families of methods. Assume for instance that the numerical method can be written in the general form

$$u_{n+1} = \sum_{j=0}^{p} a_j u_{n-j} + h \sum_{j=0}^{p} b_j f_{n-j} + h b_{-1} f_{n+1}, \; n = p, p+1, \ldots \tag{8.23}$$

for suitable coefficients $\{a_k\}$ and $\{b_k\}$ and for an integer $p \geq 0$.

Formula (8.23) defines an important family of methods, the *linear multistep methods* and $p+1$ denotes the number of steps. These methods will be analyzed with more details in Section 8.7. The initial values $u_0, u_1, \ldots, u_p$ must be provided. Apart from u_0, which is equal to y_0, the other values $u_1, \ldots, u_p$ can be generated by suitable accurate methods such as e.g., the Runge-Kutta methods that we will address in Section 8.7.

The polynomial

$$\pi(r) = r^{p+1} - \sum_{j=0}^{p} a_j r^{p-j} \tag{8.24}$$

is called the *first characteristic polynomial* associated with the numerical method (8.23), and we denote its roots by r_j, $j = 0, \ldots, p$. It can be proved that the method (8.23) is zero-stable iff the following *root condition* is satisfied:

$$\begin{cases} |r_j| \leq 1 \text{ for all } j = 0, \ldots, p, \\ \text{furthermore } \pi'(r_j) \neq 0 \text{ for those } j \text{ such that } |r_j| = 1. \end{cases} \tag{8.25}$$

For example, for the forward Euler method we have

$$p = 0, \; a_0 = 1, \; b_{-1} = 0, \; b_0 = 1,$$

for the backward Euler method we have

$$p = 0, \; a_0 = 1, \; b_{-1} = 1, \; b_0 = 0,$$

and for the Crank-Nicolson method we have

$$p = 0,\ a_0 = 1,\ b_{-1} = 1/2,\ b_0 = 1/2.$$

In all cases there is only one root of $\pi(r)$ which is equal to 1 and therefore all these methods are zero-stable.

The following property, known as Lax-Richtmyer *equivalence theorem*, is most crucial in the theory of numerical methods (see, e.g., [IK66]), and highlights the fundamental role played by the property of zero-stability:

Any consistent method is convergent iff it is zero-stable

Coherently with what done before, the local truncation error for the multistep method (8.23) is defined as follows

$$\begin{aligned}\tau_n(h) = \frac{1}{h}\Bigg\{ & y_{n+1} - \sum_{j=0}^{p} a_j y_{n-j} \\ & -h\sum_{j=0}^{p} b_j f(t_{n-j}, y_{n-j}) - hb_{-1} f(t_{n+1}, y_{n+1})\Bigg\}.\end{aligned} \tag{8.26}$$

As already noticed, the method is said to be consistent if $\tau(h) = \max|\tau_n(h)|$ tends to zero when h tends to zero. By a tedious use of Taylor expansions we can prove that this condition is equivalent to require that

$$\boxed{\sum_{j=0}^{p} a_j = 1, \qquad -\sum_{j=0}^{p} ja_j + \sum_{j=-1}^{p} b_j = 1} \tag{8.27}$$

which in turns amounts to say that $r = 1$ is a root of the polynomial $\pi(r)$ introduced in (8.24) (see, e.g., [QSS07, Chapter 11]).

8.6 Stability on unbounded intervals

In the previous section we considered the solution of the Cauchy problem on bounded intervals. In that context, the number N_h of subintervals becomes infinite only if h goes to zero.

On the other hand, there are several situations in which the Cauchy problem needs to be integrated on very large (virtually infinite) time intervals. In this case, even if h is fixed, N_h tends to infinity, and then results like (8.13) become meaningless as the right hand side of the inequality contains an unbounded quantity. We are therefore interested in methods that are able to approximate the solution for arbitrarily long time intervals, even with a steplength h relatively "large".

Unfortunately, the inexpensive forward Euler method does not enjoy this property. To see this, let us consider the following *model problem*

$$\begin{cases} y'(t) = \lambda y(t), & t \in (0,\infty), \\ y(0) = 1, \end{cases} \tag{8.28}$$

where λ is a negative real number. The exact solution is $y(t) = e^{\lambda t}$, which tends to 0 as t tends to infinity. Applying the forward Euler method to (8.28) we find that

$$u_0 = 1, \qquad u_{n+1} = u_n(1+\lambda h) = (1+\lambda h)^{n+1}, \qquad n \geq 0. \tag{8.29}$$

Thus $\lim_{n\to\infty} u_n = 0$ iff

$$\boxed{-1 < 1 + h\lambda < 1, \quad \text{i.e.} \quad h < 2/|\lambda|} \tag{8.30}$$

This condition expresses the requirement that, for *fixed* h, the numerical solution should reproduce the behavior of the exact solution when t_n tends to infinity. If $h > 2/|\lambda|$, then $\lim_{n\to\infty} |u_n| = +\infty$; thus (8.30) is a stability condition. The property that

$$\lim_{n\to\infty} u_n = 0 \tag{8.31}$$

is called *absolute stability*.

Example 8.3 Let us apply the forward Euler method to solve problem (8.28) with $\lambda = -1$. In that case we must have $h < 2$ for absolute stability. In Figure 8.4 we report the solutions obtained on the interval $[0, 30]$ for 3 different values of h: $h = 30/14$ (which violates the stability condition), $h = 30/16$ (which satisfies, although by a little amount only, the stability condition) and $h = 1/2$. We can see that in the first two cases the numerical solution oscillates. However only in the first case (which violates the stability condition) the absolute value of the numerical solution does not vanish at infinity (and actually it diverges). ■

Similar conclusions hold when λ is either a complex number (see Section 8.6.1) or when $\lambda = \lambda(t)$ in (8.28) is a negative function of t in (8.28). However in the latter case, $|\lambda|$ must be replaced by $\max_{t\in[0,\infty)} |\lambda(t)|$ in the stability condition (8.30). This condition could however be relaxed to one which is less restrictive by using a *variable steplength* $\overline{h}_n$ which accounts for the local behavior of $|\lambda(t)|$ in every interval (t_n, t_{n+1}).

In particular, the following *adaptive* forward Euler method could be used:

choose $u_0 = y_0$ and $\overline{h}_0 = 2\alpha/|\lambda(t_0)|$; then

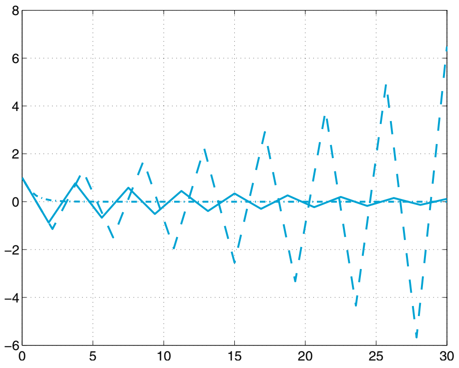


Figure 8.4. Solutions of problem (8.28), with $\lambda = -1$, obtained by the forward Euler method, corresponding to $h = 30/14$ (> 2) (*dashed line*), $h = 30/16$ (< 2) (*solid line*) and $h = 1/2$ (*dashed-dotted line*)

$$\begin{aligned}
&\text{for } n = 0, 1, \ldots, \text{ do}\\
&\qquad t_{n+1} = t_n + \overline{h}_n,\\
&\qquad u_{n+1} = u_n + \overline{h}_n \lambda(t_n) u_n,\\
&\qquad \overline{h}_{n+1} = 2\alpha/|\lambda(t_{n+1})|,
\end{aligned} \tag{8.32}$$

where α is a constant which must be less than 1 in order to have an absolutely stable method.

For instance, consider the problem

$$y'(t) = -(e^{-t} + 1)y(t), \qquad t \in (0, 10),$$

with $y(0) = 1$. Since $|\lambda(t)|$ is decreasing, the most restrictive condition for absolute stability of the forward Euler method is $h < h_0 = 2/|\lambda(0)| = 1$. In Figure 8.5, left, we compare the solution of the forward Euler method with that of the adaptive method (8.32) for three values of α. Note that, although every $\alpha < 1$ is admissible for stability purposes, to get an accurate solution requires choosing α sufficiently small. In Figure 8.5, right, we also plot the behavior of $\overline{h}_n$ on the interval $(0, 10]$ corresponding to the three values of α. This picture clearly shows that the sequence $\{\overline{h}_n\}$ increases monotonically with n.

In contrast to the forward Euler method, neither the backward Euler method nor the Crank-Nicolson method require limitations on h for absolute stability. In fact, with the backward Euler method we obtain $u_{n+1} = u_n + \lambda h u_{n+1}$ and therefore

$$u_{n+1} = \left(\frac{1}{1 - \lambda h}\right)^{n+1}, \qquad n \geq 0,$$

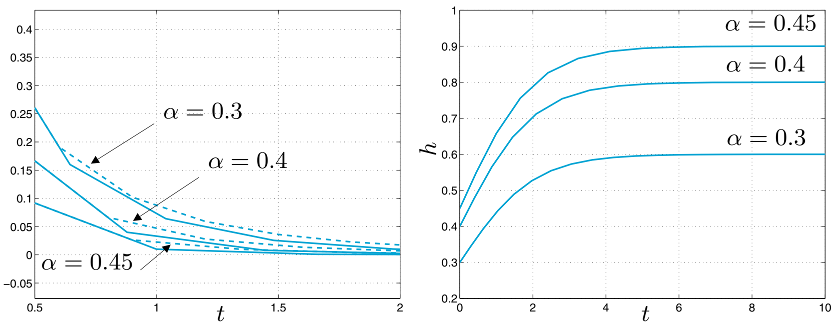


Figure 8.5. Left: the numerical solution on the time interval $(0.5, 2)$ obtained by the forward Euler method with $h = \alpha h_0$ (*dashed line*) and by the adaptive variable stepping forward Euler method (8.32) (*solid line*) for three different values of α. Right: the behavior of the variable steplength $\overline{h}_n$ for the adaptive method (8.32)

which tends to zero as $n \to \infty$ for *all values of* $h > 0$. Similarly, with the Crank-Nicolson method we obtain

$$u_{n+1} = \left[\left(1 + \frac{h\lambda}{2}\right) \Big/ \left(1 - \frac{h\lambda}{2}\right)\right]^{n+1}, \qquad n \geq 0,$$

which still tends to zero as $n \to \infty$ for all possible values of $h > 0$. We can conclude that the forward Euler method is *conditionally absolutely stable*, while both the backward Euler and Crank-Nicolson methods are *unconditionally absolutely stable.*

8.6.1 The region of absolute stability

If in (8.28) λ is a complex number with negative real part, the solution $u(t) = e^{\lambda t}$ still tends to 0 when t tends to infinity.

We call *region of absolute stability* $\mathcal{A}$ of a numerical method the set of complex numbers $z = h\lambda$ for which the method turns out to be absolutely stable (that is, $\lim_{n\to\infty} u_n = 0$).

The region of absolute stability of forward Euler method is given by those numbers $h\lambda \in \mathbb{C}$ such that $|1 + h\lambda| < 1$, thus it coincides with the circle of radius one and with centre $(-1, 0)$. This yields an upper bound $h < -2Re(\lambda)/|\lambda|^2$ for the steplength. For the backward Euler method the property of absolute stability is instead satisfied by all values of $h\lambda$ which are exterior to the circle of radius one centered in $(1, 0)$ (see Figure 8.6). Finally, the region of absolute stability of Crank-Nicolson method coincides with the left hand complex plane of numbers with negative real part.

Methods that are unconditionally absolutely stable for all complex number λ in (8.28) with negative real part are called *A-stable.* Backward

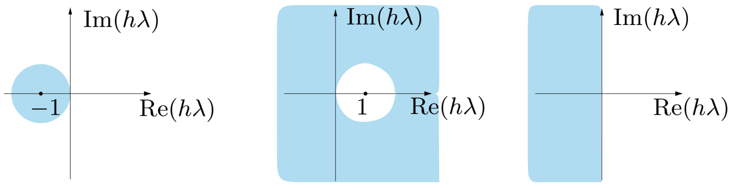


Figure 8.6. The absolute stability regions (*in cyan*) of the forward Euler method (*left*), backward Euler method (*centre*) and Crank-Nicolson method (*right*)

Euler and Crank-Nicolson method are therefore *A-stable*, and so are many other implicit methods. This property makes implicit methods attractive in spite of being computationally more expensive than explicit methods.

Example 8.4 Let us compute the restriction on h when using the forward Euler method to solve the Cauchy problem $y'(t) = \lambda y$ with $\lambda = -1 + i$. This λ stands on the boundary of the absolute stability region $\mathcal{A}$ of the forward Euler method. Thus, any h such that $h \in (0, 1)$ will suffice to guarantee that $h\lambda \in \mathcal{A}$. If it were $\lambda = -2 + 2i$ we should choose $h \in (0, 1/2)$ in order to bring $h\lambda$ within the stability region $\mathcal{A}$. ■

8.6.2 Absolute stability controls perturbations

Consider now the following *generalized model problem*

$$\begin{cases} y'(t) = \lambda(t)y(t) + r(t), & t \in (0, +\infty), \\ y(0) = 1, \end{cases} \tag{8.33}$$

where λ and r are two continuous functions and $-\lambda_{max} \le \lambda(t) \le -\lambda_{min}$ with $0 < \lambda_{min} \le \lambda_{max} < +\infty$. In this case the exact solution does not necessarily tend to zero as t tends to infinity; for instance if both r and λ are constants we have

$$y(t) = \left(1 + \frac{r}{\lambda}\right) e^{\lambda t} - \frac{r}{\lambda}$$

whose limit when t tends to infinity is $-r/\lambda$. Thus, in general, it does not make sense to require a numerical method to be absolutely stable, i.e. to satisfy (8.31), when applied to problem (8.33). However, we are going to show that a numerical method which is absolutely stable on the model problem (8.28), if applied to the generalized problem (8.33), guarantees that the perturbations are kept under control as t tends to infinity (possibly under a suitable constraint on the time-step h).

For the sake of simplicity we will confine our analysis to the forward Euler method; when applied to (8.33) it reads

$$\begin{cases} u_{n+1} = u_n + h(\lambda_n u_n + r_n), \quad n \geq 0, \\ u_0 = 1 \end{cases}$$

and its solution is (see Exercise 8.9)

$$u_n = u_0 \prod_{k=0}^{n-1}(1+h\lambda_k) + h\sum_{k=0}^{n-1} r_k \prod_{j=k+1}^{n-1}(1+h\lambda_j), \tag{8.34}$$

where $\lambda_k = \lambda(t_k)$ and $r_k = r(t_k)$, with the convention that the last product is equal to one if $k+1 > n-1$. Let us consider the following "perturbed" method

$$\begin{cases} z_{n+1} = z_n + h(\lambda_n z_n + r_n + \rho_{n+1}), \quad n \geq 0, \\ z_0 = u_0 + \rho_0, \end{cases} \tag{8.35}$$

where $\rho_0, \rho_1, \ldots$ are given perturbations which are introduced at every time level. This is a simple model in which ρ_0 and ρ_{n+1}, respectively, account for the fact that neither u_0 nor r_n can be determined exactly. (Should we account for *all* roundoff errors which are actually introduced at any step, our perturbed model would be far more involved and difficult to analyze.) The solution of (8.35) reads like (8.34), provided u_k is replaced by z_k and r_k by $r_k + \rho_{k+1}$, for all $k = 0, \ldots, n-1$. Then

$$z_n - u_n = \rho_0 \prod_{k=0}^{n-1}(1+h\lambda_k) + h\sum_{k=0}^{n-1} \rho_{k+1} \prod_{j=k+1}^{n-1}(1+h\lambda_j). \tag{8.36}$$

The quantity $|z_n - u_n|$ is called the *perturbation error* at step n. It is worth noticing that this quantity does not depend on the function $r(t)$.

i. For the sake of exposition, let us consider first the special case where λ_k and ρ_k are two constants equal to λ and ρ, respectively. Assume that $h < h_0(\lambda) = 2/|\lambda|$, which is the condition on h that ensures the absolute stability of the forward Euler method applied to the model problem (8.28). Then, using the following identity for the geometric sum

$$\sum_{k=0}^{n-1} a^k = \frac{1-a^n}{1-a}, \qquad \text{if } |a| \neq 1, \tag{8.37}$$

we obtain

$$z_n - u_n = \rho\left\{(1+h\lambda)^n\left(1+\frac{1}{\lambda}\right) - \frac{1}{\lambda}\right\}. \tag{8.38}$$

It follows that the perturbation error satisfies (see Exercise 8.10)

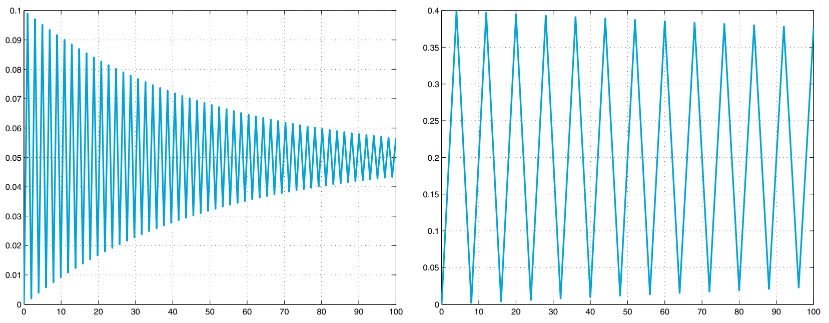


Figure 8.7. The perturbation error when $r(t) \equiv 0$, $\rho = 0.1$: $\lambda = -2$ (*left*) and $\lambda = -0.5$ (*right*). In both cases $h = h_0(\lambda) - 0.01$

$$|z_n - u_n| \leq \varphi(\lambda)|\rho|, \tag{8.39}$$

with $\varphi(\lambda) = 1$ if $\lambda \leq -1$, while $\varphi(\lambda) = |1 + 2/\lambda|$ if $-1 < \lambda < 0$. The conclusion that can be drawn is that the perturbation error is bounded by $|\rho|$ times a constant which depends on λ but is independent of both n and h. Moreover, from (8.38) it follows

$$\lim_{n\to\infty} |z_n - u_n| = \frac{|\rho|}{|\lambda|}.$$

Figure 8.7 corresponds to the case where $r(t) \equiv 0$, $\rho = 0.1$, $\lambda = -2$ (*left*) and $\lambda = -0.5$ (*right*). In both cases we have taken $h = h_0(\lambda) - 0.01$. Note that the estimate (8.38) is exactly satisfied. Obviously, the perturbation error blows up when n increases if the stability limit $h < h_0(\lambda)$ is violated.

Remark 8.3 If the unique perturbation is on the initial data, i.e. if $\rho_k = 0$, $k = 1, 2, \ldots$, from (8.36) we deduce that $\lim_{n\to\infty} |z_n - u_n| = 0$ under the stability condition $h < h_0(\lambda)$. ■

ii. In the general case where λ and r are non-constant, let us require h to satisfy the restriction $h < h_0(\lambda)$, where this time $h_0(\lambda) = 2/\lambda_{max}$. Then,

$$|1 + h\lambda_k| \leq a(h) = \max\{|1 - h\lambda_{min}|, |1 - h\lambda_{max}|\}.$$

Since $0 < \frac{\lambda_{max}-\lambda_{min}}{\lambda_{max}+\lambda_{min}} \leq a(h) < 1$, we can still use the identity (8.37) in (8.36) and obtain

$$|z_n - u_n| \leq \overline{\rho}\left([a(h)]^n + h\frac{1 - [a(h)]^n}{1 - a(h)}\right), \tag{8.40}$$

where $\overline{\rho} = \sup_k |\rho_k|$. First, let us take $h \leq h^* = 2/(\lambda_{min} + \lambda_{max})$, so that $a(h) = (1 - h\lambda_{min})$. It holds

$$|z_n - u_n| \leq \frac{\overline{\rho}}{\lambda_{min}} \left[1 - [a(h)]^n (1 - \lambda_{min})\right], \tag{8.41}$$

i.e.,

$$\sup_n |z_n - u_n| \leq \frac{\overline{\rho}}{\lambda_{min}} \sup_n [1 - [a(h)]^n (1 - \lambda_{min})].$$

If $\lambda_{min} = 1$, we have

$$\sup_n |z_n - u_n| \leq \overline{\rho}. \tag{8.42}$$

If $\lambda_{min} < 1$, the sequence $b_n = [1 - [a(h)]^n (1 - \lambda_{min})]$ monotonically increases with n, so that $\sup_n b_n = \lim_{n\to\infty} b_n = 1$ and

$$\sup_n |z_n - u_n| \leq \frac{\overline{\rho}}{\lambda_{min}}. \tag{8.43}$$

Finally, if $\lambda_{min} > 1$, the sequence b_n monotonically decreases, $\sup_n b_n = b_0 = \lambda_{min}$, and the estimate (8.42) holds too.

Let us take now $h^* < h < h_0(\lambda)$, we have

$$1 + h\lambda_k = 1 - h|\lambda_k| \leq 1 - h^*|\lambda_k| \leq 1 - h^*\lambda_{min}. \tag{8.44}$$

Using (8.44), identity (8.37) in (8.36), and setting $a = 1 - h^*\lambda_{min}$, we find

$$\begin{aligned} z_n - u_n &\leq \overline{\rho}\left(a^n + h\frac{1-a^n}{1-a}\right) \\ &= \frac{\overline{\rho}}{\lambda_{min}}\left(a^n\left(\lambda_{min} - \frac{h}{h^*}\right) + \frac{h}{h^*}\right). \end{aligned} \tag{8.45}$$

We note that two possible situations arise.
If $\lambda_{min} \geq \dfrac{h}{h^*}$, then $\dfrac{h}{h^*} \leq a^n\left(\lambda_{min} - \dfrac{h}{h^*}\right) + \dfrac{h}{h^*} < \lambda_{min}$ and we find

$$z_n - u_n \leq \overline{\rho} \qquad \forall n \geq 0. \tag{8.46}$$

Otherwise, if $\lambda_{min} < \dfrac{h}{h^*}$, then $\lambda_{min} \leq a^n\left(\lambda_{min} - \frac{h}{h^*}\right) + \frac{h}{h^*} < \frac{h}{h^*}$ and

$$z_n - u_n \leq \frac{\overline{\rho}}{\lambda_{min}}\frac{h}{h^*} \leq \frac{\overline{\rho}}{\lambda_{min}}\frac{h_0}{h^*} = \overline{\rho}\left(\frac{1}{\lambda_{min}} + \frac{1}{\lambda_{max}}\right). \tag{8.47}$$

Note that the right hand side (8.47) is also an upper bound for the absolute value of $z_n - u_n$. In Figure 8.8 we report the perturbation errors computed on the problem (8.33), where $r(t) \equiv 0$, $\lambda_k = \lambda(t_k) =$

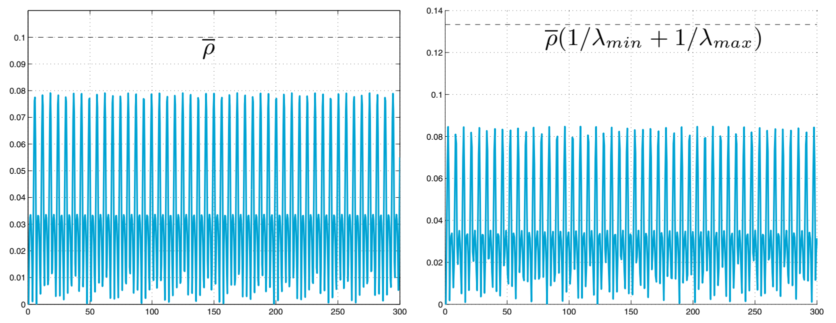


Figure 8.8. The perturbation error when $\rho(t) = 0.1\sin(t)$ and $\lambda(t) = -2 - \sin(t)$ for $t \in (0, nh)$ with $n = 500$: the steplength is $h = h^* - 0.1 = 0.4$ (*left*) and $h = h^* + 0.1 = 0.6$ (*right*). In this case $\lambda_{min} = 1$, so that the estimate (8.42) holds when $h \leq h^*$, while (8.47) holds when $h > h^*$

$-2 - \sin(t_k)$, $\rho_k = \rho(t_k) = 0.1\sin(t_k)$ with $h \leq h^*$ (*left*) and with $h^* < h < h_0(\lambda)$ (*right*).

iii. We consider now the Cauchy problem (8.5) with a general function $f(t, y(t))$. We claim that this problem can be related to the generalized model problem (8.33), in those cases where

$$-\lambda_{max} < \frac{\partial f}{\partial y}(t, y) < -\lambda_{min} \quad \forall t \geq 0,\ \forall y \in (-\infty, \infty), \tag{8.48}$$

for suitable values λ_{min}, $\lambda_{max} \in (0, +\infty)$. To this end, for every t in the generic interval (t_n, t_{n+1}), we subtract (8.6) from (8.22) to obtain the following equation for the perturbation error

$$z_n - u_n = (z_{n-1} - u_{n-1}) + h\{f(t_{n-1}, z_{n-1}) - f(t_{n-1}, u_{n-1})\} + h\rho_n.$$

By applying the mean-value theorem we obtain

$$f(t_{n-1}, z_{n-1}) - f(t_{n-1}, u_{n-1}) = \lambda_{n-1}(z_{n-1} - u_{n-1}),$$

where $\lambda_{n-1} = f_y(t_{n-1}, \xi_{n-1})$, ξ_{n-1} is a suitable point in the interval whose endpoints are u_{n-1} and z_{n-1} and f_y is a shorthand notation for $\partial f/\partial y$. Thus

$$z_n - u_n = (1 + h\lambda_{n-1})(z_{n-1} - u_{n-1}) + h\rho_n.$$

By a recursive application of this formula we obtain the identity (8.36), from which we derive the same conclusions drawn in *ii*., provided the stability restriction $0 < h < 2/\lambda_{max}$ holds. Note that this is precisely the condition (8.15).

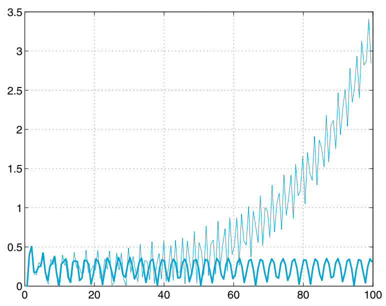


Figure 8.9. The perturbation errors when $\rho(t) = \sin(t)$ with $h = h_0 - 0.01$ (*thick line*) and $h = h_0 + 0.01$ (*thin line*) for the Cauchy problem (8.49); $h_0 = 2/3$

Example 8.5 Let us consider the Cauchy problem

$$y'(t) = \arctan(3y) - 3y + t,\ t > 0,\ y(0) = 1. \tag{8.49}$$

Since $f_y = 3/(1 + 9y^2) - 3$ is negative, we can choose $\lambda_{max} = \max|f_y| = 3$ and set $h < h_0 = 2/3$. Thus, we can expect that the perturbations on the forward Euler method are kept under control provided that $h < 2/3$. This is confirmed by the results which are reported in Figure 8.9. Note that in this example, taking $h = 2/3 + 0.01$ (thus violating the previous stability limit) the perturbation error blows up as t increases. ■

Example 8.6 We seek an upper bound on h that guarantees stability for the forward Euler method applied to approximate the Cauchy problem

$$y' = 1 - y^2, \qquad t > 0, \tag{8.50}$$

with $y(0) = \dfrac{e-1}{e+1}$. The exact solution is $y(t) = (e^{2t+1} - 1)/(e^{2t+1} + 1)$ and $f_y = -2y$. Since $f_y \in (-2, -0.9)$ for all $t > 0$, we can take h less than $h_0 = 1$. In Figure 8.10, left, we report the solutions obtained on the interval $(0, 35)$ with $h = 0.95$ (*thick line*) and $h = 1.05$ (*thin line*). In both cases the solution oscillates, but remains bounded. Moreover in the first case, which satisfies the stability constraint, the oscillations are damped and the numerical solution tends to the exact one as t increases. In Figure 8.10, right, we report the perturbation errors corresponding to $\rho(t) = \sin(t)$ with $h = h^* = 2/2.9$ (*thick solid line*) and $h = 0.9$ (*thin dashed line*). In both cases the perturbation errors remain bounded; precisely, estimate (8.42) is satisfied when $h = h^* = 2/2.9$, while estimate (8.47) holds when $h^* < h = 0.9 < h_0$. ■

In those cases where no information on y is available, finding the value $\lambda_{max} = \max|f_y|$ is not a simple matter. A more heuristic approach could be pursued in these situations, by adopting a variable stepping procedure. Precisely, one could take $t_{n+1} = t_n + h_n$, where

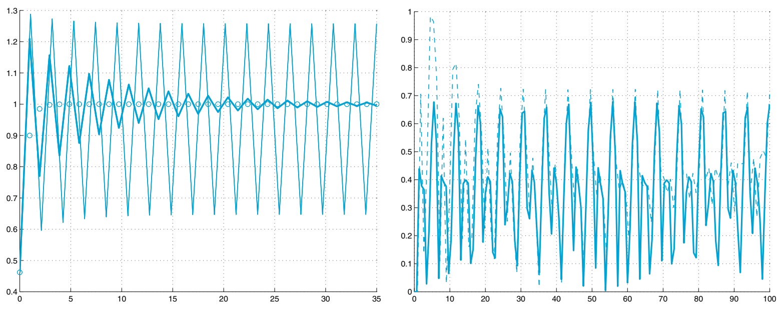


Figure 8.10. At left, numerical solutions of problem (8.50) obtained by the forward Euler method with $h = 1.05$ (*thin line*) and $h = 0.95$ (*thick line*). The values of the exact solution are indicated by circles. On the right, perturbation errors corresponding to $\rho(t) = \sin(t)$ with $h = h^* = 2/2.9$ (*thick solid line*) and $h = 0.9$ (*thin dashed line*)

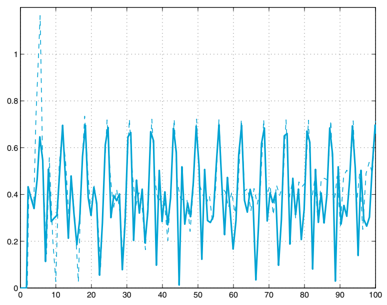


Figure 8.11. The perturbation errors corresponding to $\rho(t) = \sin(t)$ with $\alpha = 0.8$ (*thick line*) and $\alpha = 0.9$ (*thin line*) for the Example 8.6, using the adaptive strategy

$$0 < h_n < 2\frac{\alpha}{|f_y(t_n, u_n)|}, \tag{8.51}$$

for suitable values of α strictly less than 1. Note that the denominator depends on the value u_n which is known. In Figure 8.11 we report the perturbation errors corresponding to the Example 8.6 for two different values of α.

The previous analysis can be carried out also for other kind of one-step methods, in particular for the backward Euler and Crank-Nicolson methods. For these methods which are A-stable, the same conclusions about the perturbation error can be drawn without requiring any limitation on the time-step. In fact, in the previous analysis one should replace each term $1 + h\lambda_n$ by $(1 - h\lambda_n)^{-1}$ in the backward Euler case and by $(1 + h\lambda_n/2)/(1 - h\lambda_n/2)$ in the Crank-Nicolson case.

8.6.3 Stepsize adaptivity for the forward Euler method

As seen in the previous sections, the steplength h should be chosen in order to satisfy the absolute stability constraint, see e.g. (8.32) and (8.51).

More in general, at every time level we could in principle choose a (variable) time-step that not only fulfils the stability constraint but also guarantees that a desired accuracy be achieved. Such procedure, called *step adaptivity*, requires a convenient estimate of the local error, that is obtained from an appropriate a-posteriori error estimate. (A priori error estimates like (8.13) or (8.14) do not serve this porpuse, as they would require information on the second derivative of the unknown solution.) For the sake of simplicity, we illustrate this technique on the forward Euler method.

Assume that the numerical solution is computed up to a given time level that, for simplicity, will be denoted $\overline{t}$. We choose an initial guess for h and denote by u_h (respectively, $u_{h/2}$) the solution at the time $\overline{t}+h$ provided by the forward Euler method with initial value $\overline{u}$ at time $\overline{t}$ with time-step h (respectively, $h/2$), that is:

$$\begin{aligned} u_h &= \overline{u} + hf(\overline{t},\overline{u}), \\ v_1 &= \overline{u} + \frac{h}{2}f(\overline{t},\overline{u}), \qquad u_{h/2} = v_2 = v_1 + \frac{h}{2}f\left(\overline{t}+\frac{h}{2}, v_1\right). \end{aligned}$$

Let us examine the errors $e_h = y(\overline{t}+h) - u_h$ and $e_{h/2} = y(\overline{t}+h) - u_{h/2}$, where now $y(t)$ represents the exact solution to the Cauchy problem

$$\begin{cases} y'(t) = f(t,y(t)) & t \geq \overline{t}, \\ y(\overline{t}) = \overline{u}. \end{cases} \tag{8.52}$$

Using (8.10) we find

$$e_h = \frac{h^2}{2}y''(\xi) \tag{8.53}$$

for a suitable $\xi \in (\overline{t}, \overline{t}+h)$. By setting, for simplicity,

$$t_0 = \overline{t}, \qquad t_1 = \overline{t}+h/2, \qquad t_2 = \overline{t}+h$$

(see Figure 8.12) and rewriting $e_{h/2}$ in the form (8.9), we find

$$e_{h/2} = (y(t_2) - v_2^*) + (v_2^* - v_2), \tag{8.54}$$

where $v_2^* = y(t_1) + \frac{h}{2}f(t_1, y(t_1))$. The former term on the right hand side of (8.54) represents the local truncation error, thus

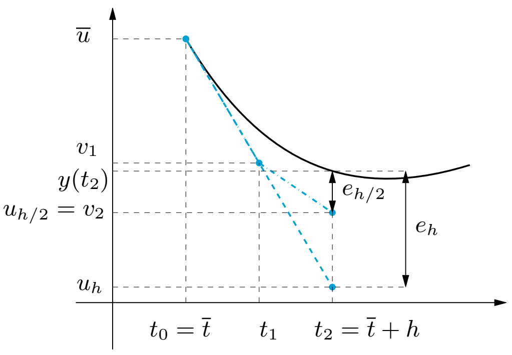


Figure 8.12. The numerical solution provided by forward Euler method with either one step of size h and two steps of size $h/2$. The solid curve represents the solution of (8.52)

$$y(t_2) - v_2^* = \frac{(h/2)^2}{2} y''(\eta_2)$$

for a suitable $\eta_2 \in (t_1, t_2)$, whereas the latter, that is due to the error propagation on an interval of length $h/2$, thanks to (8.12) reads

$$v_2^* - v_2 = (y(t_1) - v_1) + \frac{h}{2}\left[f(t_1, y(t_1)) - f(t_1, v_1)\right].$$

The term $(y(t_1) - v_1)$ still represents a local truncation error which can be written as $y(t_1) - v_1 = \dfrac{(h/2)^2}{2} y''(\eta_1)$ for a suitable $\eta_1 \in (t_0, t_1)$. On the other hand, assuming f of class C^1 and using the Lagrange theorem, we obtain

$$f(t_1, y(t_1)) = f(t_1, v_1) + (y(t_1) - v_1)\frac{\partial f}{\partial y}(t_1, \zeta)$$

for a suitable ζ belonging to the interval whose endpoints are v_1 and $y(t_1)$. Consequently

$$v_2^* - v_2 = (y(t_1) - v_1)\left[1 + h\frac{\partial f}{\partial y}(t_1, \zeta)\right] = \frac{(h/2)^2}{2} y''(\eta_1) + o(h^2).$$

Assuming that y'' is continuous in $(\overline{t}, \overline{t} + h)$, we have

$$e_{h/2} = \frac{(h/2)^2}{2}\left[y''(\eta_2) + y''(\eta_1)\right] + o(h^2) = \frac{h^2}{4} y''(\eta) + o(h^2), \qquad (8.55)$$

for a suitable $\eta \in (\overline{t}, \overline{t} + h)$.

A convenient estimate of y'' can be obtained by subtracting (8.55) from (8.53). Still assuming that y'' is continuous in $(\overline{t}, \overline{t} + h)$, we find

$$u_{h/2} - u_h = e_h - e_{h/2} = \frac{h^2}{4}\left(2y''(\xi) - y''(\eta)\right) + o(h^2) = \frac{h^2}{4} y''(\hat{\xi}) + o(h^2),$$

for a convenient $\hat{\xi} \in (\bar{t}, \bar{t}+h)$. On the other hand

$$|e_{h/2}| \simeq \frac{h^2}{4}|y''(\hat{\xi})| \simeq |u_{h/2} - u_h|,$$

therefore the quantity $|u_{h/2} - u_h|$ provides an a-posteriori estimator of the error $|y(\bar{t}+h) - u_{h/2}|$ up to an infinitesimal term $o(h^2)$.

To conclude, for a given tolerance ϵ, should

$$|u_{h/2} - u_h| < \frac{\epsilon}{2}$$

(the division by 2 is made conservatively), we accept the step h to advance and take $u_{h/2}$ as our numerical solution at the new time level $\bar{t}+h$. Otherwise, h is halved and the above procedure is repeated until convergence. In any case, to avoid too tiny steplengths we require that the steplength satisfies $h \geq h_{min}$ for a prescribed minimum value h_{min}.

We finally observe that sometimes the error estimator $|u_{h/2} - u_h|$ is replaced by its relative counterpart $|u_{h/2} - u_h|/u_{max}$, where u_{max} represents the maximum value attained by the numerical solution in the interval $[t_0, \bar{t}]$.

Let us summarize

1. An absolutely stable method is one which generates a solution u_n of the model problem (8.28) which tends to zero as t_n tends to infinity;
2. a method is said *A-stable* if it is absolutely stable for any possible choice of the time-step (or steplength) h and any $\lambda \in \mathbb{C}$ with $Re(\lambda) < 0$ (otherwise a method is called conditionally stable, and h should be lower than a constant depending on λ);
3. when an absolutely stable method is applied to a generalized model problem (like (8.33)), the perturbation error (that is the absolute value of the difference between the perturbed and unperturbed solution) is uniformly bounded with respect to h. In short, we can say that absolutely stable methods keep the perturbation controlled;
4. the analysis of absolute stability for the linear model problem can be exploited to find stability conditions on the time-step when considering the nonlinear Cauchy problem (8.5) with a function f satisfying (8.48). In that case the stability restriction requires the steplength to be chosen as a function of $\partial f/\partial y$. Precisely, the new integration interval $[t_n, t_{n+1}]$ is chosen in such a way that $h_n = t_{n+1} - t_n$ satisfies (8.51) for a suitable $\alpha \in (0,1)$, or (8.15) in the case of constant time-step h.

See the Exercises 8.6-8.13.

8.7 High order methods

All methods presented so far are elementary examples of one-step methods. More sophisticated schemes, which allow the achievement of a higher order of accuracy, are the *Runge-Kutta methods* and the *multistep methods* (whose general form was already introduced in (7.23)).

Runge-Kutta (briefly, RK) methods are still one-step methods; however, they involve several evaluations of the function $f(t, y)$ on every interval $[t_n, t_{n+1}]$. In its most general form, a RK method can be written as

$$u_{n+1} = u_n + h\sum_{i=1}^{s} b_i K_i, \qquad n \geq 0 \tag{8.56}$$

where

$$K_i = f(t_n + c_i h, u_n + h\sum_{j=1}^{s} a_{ij} K_j), \quad i = 1, 2, \ldots, s$$

and s denotes the number of *stages* of the method. The coefficients $\{a_{ij}\}$, $\{c_i\}$ and $\{b_i\}$ fully characterize a RK method and are usually collected in the so-called *Butcher array*

$$\begin{array}{c|c} \mathbf{c} & \mathrm{A} \\ \hline & \mathbf{b}^T \end{array},$$

where $\mathrm{A} = (a_{ij}) \in \mathbb{R}^{s\times s}$, $\mathbf{b} = (b_1, \ldots, b_s)^T \in \mathbb{R}^s$ and $\mathbf{c} = (c_1, \ldots, c_s)^T \in \mathbb{R}^s$. If the coefficients a_{ij} in A are equal to zero for $j \geq i$, with $i = 1, 2, \ldots, s$, then each K_i can be explicitly computed in terms of the $i-1$ coefficients $K_1, \ldots, K_{i-1}$ that have already been determined. In such a case the RK method is *explicit*.
Otherwise, it is *implicit* and solving a nonlinear system of size s is necessary for computing the coefficients K_i.

One of the most celebrated Runge-Kutta methods reads

$$\boxed{u_{n+1} = u_n + \frac{h}{6}(K_1 + 2K_2 + 2K_3 + K_4)} \tag{8.57}$$

where

$$\begin{aligned} K_1 &= f_n, \\ K_2 &= f(t_n + \tfrac{h}{2}, u_n + \tfrac{h}{2}K_1), \\ K_3 &= f(t_n + \tfrac{h}{2}, u_n + \tfrac{h}{2}K_2), \\ K_4 &= f(t_{n+1}, u_n + hK_3), \end{aligned} \qquad \begin{array}{c|cccc} 0 & & & & \\ \frac{1}{2} & \frac{1}{2} & & & \\ \frac{1}{2} & 0 & \frac{1}{2} & & \\ 1 & 0 & 0 & 1 & \\ \hline & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} \end{array}.$$

This method can be derived from (8.18) by using the Simpson quadrature rule (4.23) to evaluate the integral between t_n and t_{n+1}. It is explicit, of fourth order with respect to h; at each time level, it involves four new evaluations of the function f. Other Runge-Kutta methods, either explicit or implicit, with arbitrary order can be constructed. For instance, an implicit RK method of order 4 with 2 stages is defined by the following Butcher array

$$\begin{array}{c|cc} \frac{3-\sqrt{3}}{6} & \frac{1}{4} & \frac{3-2\sqrt{3}}{12} \\ \frac{3+\sqrt{3}}{6} & \frac{3+2\sqrt{3}}{12} & \frac{1}{4} \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array} .$$

The absolute stability region $\mathcal{A}$ of the RK methods, including explicit RK methods, can grow in surface with the order: an example is provided by the left graph in Figure 8.14, where $\mathcal{A}$ has been reported for some explicit RK methods of increasing order: RK1, i.e. the forward Euler method; RK2, the so called *improved Euler method* that will be derived later (see (8.64)); RK3, the method associated with the following Butcher array

$$\begin{array}{c|ccc} 0 & & & \\ \frac{1}{2} & \frac{1}{2} & & \\ 1 & -1 & 2 & \\ \hline & \frac{1}{6} & \frac{2}{3} & \frac{1}{6} \end{array} \qquad (8.58)$$

and RK4, the method (8.57) introduced previously.

As done for the forward Euler method, also RK method, as one-step methods, are well-suited for implementing a steplength adaptivity.
The error estimator for these methods can be constructed in two ways:
- using the same RK method, but with two different steplengths (as done for the Euler method);
- using two RK methods of different order, but with the same number s of stages.
The latter procedure is the one used by MATLAB inside the functions `ode23` and `ode45`, see below.

RK methods stand at the base of a family of MATLAB programs whose names contain the root `ode` followed by numbers and letters. In particular, `ode45` is based on a pair of explicit Runge-Kutta methods (the so-called Dormand-Prince pair) of order 4 and 5, respectively. `ode23` is the implementation of another pair of explicit Runge-Kutta methods (the Bogacki and Shampine pair). In these methods the integration step varies in order to guarantee that the error remains below a given tolerance (the default scalar relative error tolerance `RelTol` is equal to 10^{-3}). The program `ode23tb` is an implementation of an implicit Runge-Kutta

`ode`
`ode45`
`ode23`
`ode23tb`

formula whose first stage is the trapezoidal rule, while the second stage is a backward differentiation formula of order two (see (8.61)).

Multistep methods (see (8.23)) achieve a high order of accuracy by involving the values $u_n, u_{n-1}, \ldots, u_{n-p}$ for the determination of u_{n+1}. They can be derived by applying first the formula (8.18) and then approximating the integral by a quadrature formula which involves the interpolant of f at a suitable set of nodes. A notable example of multistep method is the three-step ($p = 2$), third order (explicit) Adams-Bashforth formula (AB3)

$$\boxed{u_{n+1} = u_n + \frac{h}{12}(23f_n - 16f_{n-1} + 5f_{n-2})} \tag{8.59}$$

which is obtained by replacing f in (8.18) by its interpolating polynomial of degree two at the nodes t_{n-2}, t_{n-1}, t_n. Another important example is the three-step, fourth order (implicit) Adams-Moulton formula (AM4)

$$\boxed{u_{n+1} = u_n + \frac{h}{24}(9f_{n+1} + 19f_n - 5f_{n-1} + f_{n-2})} \tag{8.60}$$

which is obtained by replacing f in (8.18) by its interpolating polynomial of degree three at the nodes $t_{n-2}, t_{n-1}, t_n, t_{n+1}$.

Another family of multistep methods can be obtained by writing the differential equation at time t_{n+1} and replacing $y'(t_{n+1})$ by a one-sided incremental ratio of high order. An instance is provided by the two-step, second order (implicit) *backward difference formula* (BDF2)

$$\boxed{u_{n+1} = \frac{4}{3}u_n - \frac{1}{3}u_{n-1} + \frac{2h}{3}f_{n+1}} \tag{8.61}$$

or by the following three-step, third order (implicit) *backward difference formula* (BDF3)

$$\boxed{u_{n+1} = \frac{18}{11}u_n - \frac{9}{11}u_{n-1} + \frac{2}{11}u_{n-2} + \frac{6h}{11}f_{n+1}} \tag{8.62}$$

All these methods can be recasted in the general form (8.23). It is easy to verify that for all of them the relations (8.27) are satisfied, thus these methods are consistent. Moreover, they are zero-stable. Indeed, in both cases (8.59) and (8.60), the first characteristic polynomial is $\pi(r) = r^3 - r^2$ and its roots are $r_0 = 1$, $r_1 = r_2 = 0$; that of (8.61) is $\pi(r) = r^2 - (4/3)r + 1/3$ and its roots are $r_0 = 1$ and $r_1 = 1/3$, while the first characteristic polynomial of (8.62) is $\pi(r) = r^3 - 18/11r^2 + 9/11r - 2/11$ and its roots are $r_0 = 1$, $r_1 = 0.3182 + 0.2839i$, $r_2 = 0.3182 - 0.2839i$,

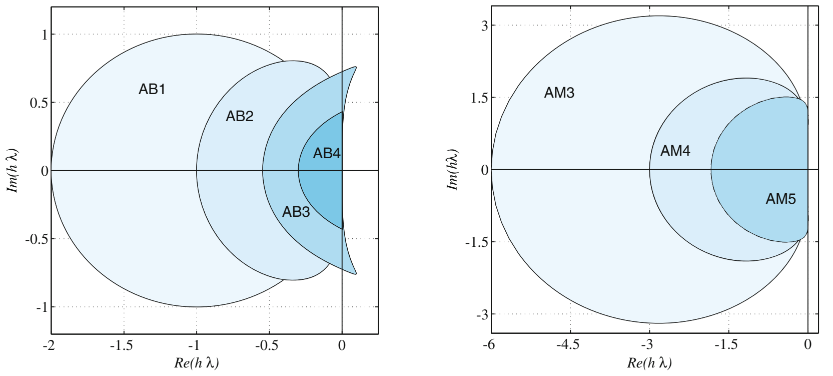


Figure 8.13. The absolute stability regions of several Adams-Basforth (*left*) and Adams-Moulton (*right*) methods

where i is the imaginary unit. In all cases, the root condition (8.25) is satisfied.

When applied to the model problem (8.28), for any $\lambda \in \mathbb{R}^-$ the method AB3 is absolutely stable if $h < 0.545/|\lambda|$, while AM4 is absolutely stable if $h < 3/|\lambda|$. The method BDF2 is unconditionally absolutely stable for any $\lambda \in \mathbb{C}$ with negative real part (i.e., A-stable). If $\lambda \in \mathbb{R}^-$, BDF3 is unconditionally absolutely stable, however this is no longer true for any $\lambda \in \mathbb{C}$ with negative real part; in other words, BDF3 fails to be A-stable (see, Figure 8.14). More generally, according to the second Dahlquist barrier there is no multistep A-stable method of order strictly greater than two.

In Figures 8.13 the regions of absolute stability of several Adams-Bashfort and Adams-Moulton methods are drawn. Note that their size reduces as far as the order increases. In the right-hand side graphs of Figure 8.14 we report the (unbounded) absolute stability regions of some BDF methods: note that $\mathcal{A}_{BDF(k+1)} \subset \mathcal{A}_{BDF(k)}$ as opposed to those of the Runge-Kutta methods (reported on the left) which instead increase in surface when the order increases, that is $\mathcal{A}_{RK(k)} \subset \mathcal{A}_{RK(k+1)}$, $k \geq 1$.

Remark 8.4 (How to draw absolute stability regions) The boundary $\partial\mathcal{A}$ of the absolute stability region $\mathcal{A}$ of a multistep method can be regarded as the set of the complex numbers $h\lambda$ such that

$$h\lambda = \left(r^{p+1} - \sum_{j=0}^{p} a_j r^{p-j} \right) \Big/ \left(\sum_{j=-1}^{p} b_j r^{p-j} \right), \tag{8.63}$$

where r is a complex number of modulus equal to one. An approximation of $\partial\mathcal{A}$ can be obtained by evaluating the right hand side of (8.63) for different values of r on the unit circle (for instance, by setting `r = exp(i*pi*(0:2000)/1000)`, where `i` is the imaginary unit). The graphs in Figures 8.13 and 8.14 have indeed been obtained in this way. ■

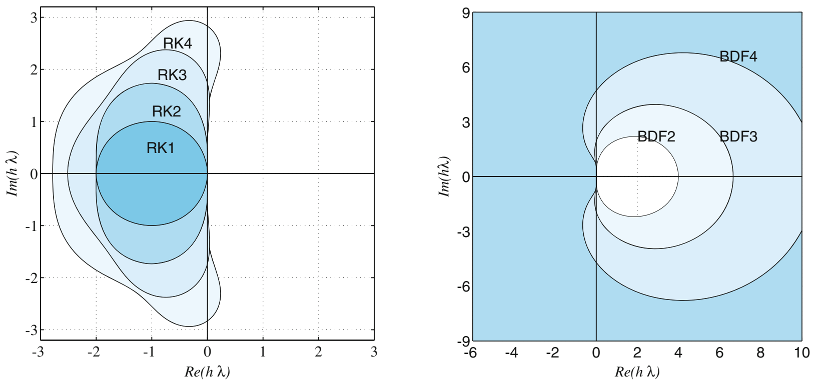


Figure 8.14. The absolute stability regions of several explicit RK (*left*) and BDF methods (*right*). In the latter case the stability regions are unbounded and they spread outside the closed curves

According to the first Dahlquist barrier the maximum order q of a $p+1$-step method satisfying the root condition is $q = p+1$ for explicit methods and, for implicit methods $q = p+2$ if $p+1$ is odd, $q = p+3$ if $p+1$ is even.

Remark 8.5 (Cyclic composite methods) Dahlquist barriers can be overcome by appropriately combining several multistep methods. For instance, the two following methods

$$
\begin{aligned}
u_{n+1} &= -\frac{8}{11}u_n + \frac{19}{11}u_{n-1} + \frac{h}{33}(30f_{n+1} + 57f_n + 24f_{n-1} - f_{n-2}),\\
u_{n+1} &= \frac{449}{240}u_n + \frac{19}{30}u_{n-1} - \frac{361}{240}u_{n-2}\\
&\quad + \frac{h}{720}(251f_{n+1} + 456f_n - 1347f_{n-1} - 350f_{n-2}),
\end{aligned}
$$

have order five, but are both unstable. However, combined in such a way that the former is used for n even, the latter for n odd, they give rise to an A-stable 3-step method of order five. ■

ode15s

Multistep methods are implemented in several MATLAB programs, for instance in `ode15s`.

Octave 8.1 `ode23` and `ode45` are also available in Octave-forge. The optional arguments however differ from MATLAB. Note that `ode45` in Octave-forge offers two possible strategies: the default one based on the Dormand and Prince method generally produces more accurate results than the other option that is based on the Fehlberg method. The built-in ODE and DAE (Differential Algebraic Equations) solvers in Octave (`lsode`, `daspk`, `dassl`, not available in MATLAB) also use multistep methods, in particular lsode can use either Adams or BDF formulas while `dassl` and `daspk` use BDF formulas. ■

8.8 The predictor-corrector methods

In Section 8.3 it was pointed out that if the function f of Cauchy problem is nonlinear, implicit methods yield at each step a nonlinear problem for the unknown value u_{n+1}. For its solution we can use one of the methods introduced in Chapter 2, or else apply the function `fsolve` as we have done with the Programs 8.2 and 8.3.

Alternatively, we can carry out fixed point iterations at every time level. For example, for the Crank-Nicolson method (8.17), for $k = 0, 1, \ldots$, we compute until convergence

$$u_{n+1}^{(k+1)} = u_n + \frac{h}{2}\left[f_n + f(t_{n+1}, u_{n+1}^{(k)})\right].$$

It can be proved that if the initial guess $u_{n+1}^{(0)}$ is chosen conveniently, a single iteration suffices in order to obtain a numerical solution $u_{n+1}^{(1)}$ whose accuracy is of the same order as the solution u_{n+1} of the original implicit method. More precisely, if the original implicit method has order $p \geq 2$, then the initial guess $u_{n+1}^{(0)}$ must be generated by an explicit method of order (at least) $p - 1$.

For instance, if we use the first order (explicit) forward Euler method to initialize the Crank-Nicolson method, we get the *Heun method* (also called *improved Euler method*), already referred as RK2:

$$\boxed{\begin{aligned} &u_{n+1}^* = u_n + hf_n,\\ &u_{n+1} = u_n + \frac{h}{2}\left[f_n + f(t_{n+1}, u_{n+1}^*)\right] \end{aligned}} \tag{8.64}$$

The explicit step is called a *predictor*, whereas the implicit one is called a *corrector*. Another example combines the (AB3) method (8.59) as predictor with the (AM4) method (8.60) as corrector. These kinds of methods are therefore called *predictor-corrector* methods. They enjoy the order of accuracy of the corrector method. However, being explicit, they undergo a stability restriction which is typically the same as that of the predictor method (see, for instance, the regions of absolute stability of Figure 8.15). Thus they are not adequate to integrate a Cauchy problem on unbounded intervals.

In Program 8.4 we implement a general predictor-corrector method. The function handles `predictor` and `corrector` identify the type of method that is chosen. For instance, if we use the functions `feonestep` and `cnonestep`, which are defined in Programs 8.5 and 8.7, respectively, we can call `predcor` as follows

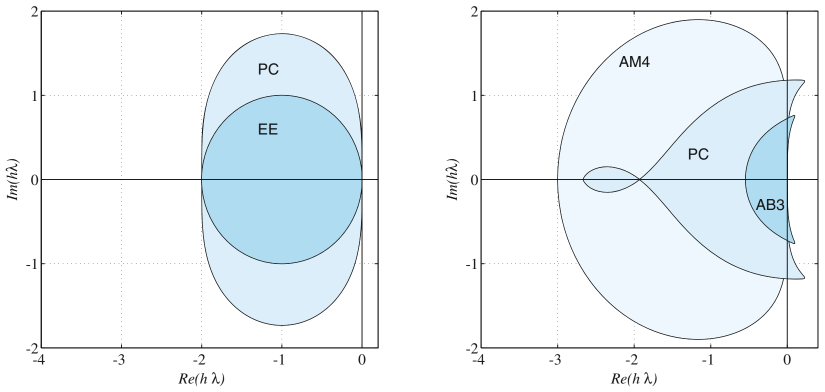


Figure 8.15. The absolute stability regions of the predictor-corrector (PC) methods obtained by combining the explicit Euler (EE) and Crank-Nicolson methods (*left*) and AB3 and AM4 (*right*). Notice the reduced surface of the region when compared to the corresponding implicit methods (in the first case the region of the Crank-Nicolson method hasn't been reported as it coincides with all the complex half-plane $Re(h\lambda) < 0$)

```
[t,u]=predcor(f,[t0,T],y0,N,@feonestep,@cnonestep);
```

and obtain the Heun method.

Program 8.4. predcor: predictor-corrector method

```
function [t,u]=predcor(odefun,tspan,y0,Nh,...
               predictor,corrector,varargin)
%PREDCOR  Solves differential equations using a
%  predictor-corrector method
%  [T,Y]=PREDCOR(ODEFUN,TSPAN,Y0,NH,PRED,CORR) with
%  TSPAN=[T0 TF] integrates the system of differential
%  equations y' = f(t,y) from time T0 to TF with
%  initial condition Y0 using a general predictor
%  corrector method on an equispaced grid of NH steps.
%  Function ODEFUN(T,Y) must return a vector, whose
%  elements hold the evaluation of f(t,y), of the
%  same dimension of Y.
%  Each row in the solution array Y corresponds to a
%  time returned in the  column vector T.
%  [T,Y]=PREDCOR(ODEFUN,TSPAN,Y0,NH,PRED,CORR,P1,..)
%  passes the additional parameters P1,... to the
%  functions ODEFUN,PRED and CORR as ODEFUN(T,Y,P1,..),
%  PRED(T,Y,P1,P2...), CORR(T,Y,P1,P2...).
h=(tspan(2)-tspan(1))/Nh;
y=y0(:); w=y; u=y.';
tt=linspace(tspan(1),tspan(2),Nh+1);
for t=tt(1:end-1)
   fn = odefun(t,w,varargin{:});
   upre = predictor(t,w,h,fn);
   w = corrector(t+h,w,upre,h,odefun,...
          fn,varargin{:});
```

```
    u = [u; w.'];
end
t = tt';
end
```

Program 8.5. feonestep: one step of the forward Euler method

```
function [u]=feonestep(t,y,h,f)
% FEONESTEP one step of the forward Euler method
u = y + h*f;
return
```

Program 8.6. beonestep: one step of the backward Euler method

```
function [u]=beonestep(t,u,y,h,f,fn,varargin)
% BEONESTEP  one step of the backward Euler method
u = u + h*f(t,y,varargin{:});
return
```

Program 8.7. cnonestep: one step of the Crank-Nicolson method

```
function [u]=cnonestep(t,u,y,h,f,fn,varargin)
% CNONESTEP one step of the Crank-Nicolson method
u = u + 0.5*h*(f(t,y,varargin{:})+fn);
return
```

The MATLAB program `ode113` implements a combined Adams-Bashforth-Moulton scheme with variable steplength. `ode113`

See the Exercises 8.14-8.17.

8.9 Systems of differential equations

Let us consider the following system of first-order ordinary differential equations whose unknowns are $y_1(t), \ldots, y_m(t)$:

$$\begin{cases} y_1' = f_1(t, y_1, \ldots, y_m), \\ \vdots \\ y_m' = f_m(t, y_1, \ldots, y_m), \end{cases}$$

where $t \in (t_0, T]$, with the initial conditions

$$y_1(t_0) = y_{0,1}, \; \ldots, \; y_m(t_0) = y_{0,m}.$$

For its solution we could apply to each individual equation one of the methods previously introduced for a scalar problem. For instance, the nth step of the forward Euler method would read

$$\begin{cases} u_{n+1,1} = u_{n,1} + hf_1(t_n, u_{n,1}, \ldots, u_{n,m}), \\ \vdots \\ u_{n+1,m} = u_{n,m} + hf_m(t_n, u_{n,1}, \ldots, u_{n,m}). \end{cases}$$

By writing the system in vector form $\mathbf{y}'(t) = \mathbf{F}(t, \mathbf{y}(t))$, with obvious choice of notation, the extension of the methods previously developed for the case of a single equation to the vector case is straightforward. For instance, the method

$$\mathbf{u}_{n+1} = \mathbf{u}_n + h(\vartheta\mathbf{F}(t_{n+1}, \mathbf{u}_{n+1}) + (1-\vartheta)\mathbf{F}(t_n, \mathbf{u}_n)), \qquad n \geq 0,$$

with $\mathbf{u}_0 = \mathbf{y}_0$, $0 \leq \vartheta \leq 1$, is the vector form of the forward Euler method if $\vartheta = 0$, the backward Euler method if $\vartheta = 1$ and the Crank-Nicolson method if $\vartheta = 1/2$.

Example 8.7 (Population dynamics) Let us apply the forward Euler method to solve the Lotka-Volterra equations (8.3) with $C_1 = C_2 = 1$, $b_1 = b_2 = 0$ and $d_1 = d_2 = 1$. In order to use Program 8.1 for a *system* of ordinary differential equations, let us create a function `f` which contains the component of the vector function $\mathbf{F}$, which we save in the file `f.m`. For our specific system we have:

```
function fn = f(t,y,C1,C2,d1,d2,b1,b2)
[n,m]=size(y); fn=zeros(n,m);
fn(1)=C1*y(1)*(1-b1*y(1)-d2*y(2));
fn(2)=-C2*y(2)*(1-b2*y(2)-d1*y(1));
return
```

Now we execute Program 8.1 with the following instructions

```
C1=1; C2=1; d1=1; d2=1; b1=0; b2=0;
[t,u]=feuler(@f,[0,10],[2 2],20000,C1,C2,d1,d2,b1,b2);
```

They correspond to solving the Lotka-Volterra system on the time interval $[0, 10]$ with a time-step $h = 5 \cdot 10^{-4}$.

The graph in Figure 8.16, left, represents the time evolution of the two components of the solution. Note that they are periodic. The graph in Figure 8.16, right, shows the trajectory issuing from the initial value in the so-called *phase plane*, that is, the Cartesian plane whose coordinate axes are y_1 and y_2. This trajectory is confined within a bounded region of the (y_1, y_2) plane. If we start from the point $(1.2, 1.2)$, the trajectory would stay in an even smaller region surrounding the point $(1, 1)$. This can be explained as follows. Our differential system admits 2 *points of equilibrium* at which $y_1' = 0$ and $y_2' = 0$, and one of them is precisely $(1, 1)$ (the other being $(0, 0)$). Actually, they are obtained by solving the nonlinear system

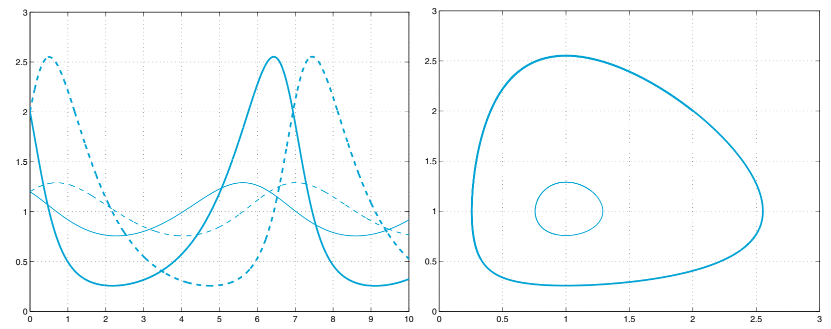


Figure 8.16. Numerical solutions of system (8.3). At left, we represent y_1 and y_2 on the time interval $(0, 10)$, the solid line refers to y_1, the dashed line to y_2. Two different initial data are considered: $(2, 2)$ (*thick lines*) and $(1.2, 1.2)$ (*thin lines*). At right, we report the corresponding trajectories in the phase plane

$$\begin{cases} y_1' = y_1 - y_1 y_2 = 0, \\ \\ y_2' = -y_2 + y_2 y_1 = 0. \end{cases}$$

If the initial data coincide with one of these points, the solution remains constant in time. Moreover, while $(0, 0)$ is an unstable equilibrium point, $(1, 1)$ is stable, that is, all trajectories issuing from a point near $(1, 1)$ stay bounded in the phase plane. ■

When we use an explicit method, the steplength h should undergo a stability restriction similar to the one encountered in Section 8.6. When the real part of the eigenvalues λ_k of the Jacobian $\mathrm{A}(t) = [\partial \mathbf{F}/\partial \mathbf{y}](t, \mathbf{y})$ of $\mathbf{F}$ are all negative, we can set $\lambda = -\max_t \rho(\mathrm{A}(t))$, where $\rho(\mathrm{A}(t))$ is the spectral radius of $\mathrm{A}(t)$. This λ is a candidate to replace the one entering in the stability conditions (such as, e.g., (8.30)) that were derived for the scalar Cauchy problem.

Remark 8.6 The MATLAB programs (`ode23`, `ode45`, ...) that we have mentioned before can be used also for the solution of systems of ordinary differential equations. The syntax is `odeXX(@f,[t0 tf],y0)`, where `y0` is the vector of the initial conditions, `f` is a function to be specified by the user and `odeXX` is one of the methods available in MATLAB. ■

Now consider the case of an ordinary differential equation of order m

$$y^{(m)}(t) = f(t, y, y', \ldots, y^{(m-1)}) \tag{8.65}$$

for $t \in (t_0, T]$, whose solution (when existing) is a family of functions defined up to m arbitrary constants. The latter can be fixed by prescribing m initial conditions

$$y(t_0) = y_0,\ y'(t_0) = y_1,\ \ldots,\ y^{(m-1)}(t_0) = y_{m-1}.$$

Setting

$$w_1(t) = y(t),\, w_2(t) = y'(t),\, \ldots,\, w_m(t) = y^{(m-1)}(t),$$

the equation (8.65) can be transformed into a first-order system of m differential equations

$$\begin{cases} w_1' = w_2, \\ w_2' = w_3, \\ \vdots \\ w_{m-1}' = w_m, \\ w_m' = f(t, w_1, \ldots, w_m), \end{cases}$$

with initial conditions

$$w_1(t_0) = y_0,\, w_2(t_0) = y_1,\, \ldots,\, w_m(t_0) = y_{m-1}.$$

Thus we can always approximate the solution of a differential equation of order $m > 1$ by resorting to the equivalent system of m first-order equations, and then applying to this system a convenient discretization method.

Example 8.8 (Electrical circuits) Consider the circuit of Problem 8.4 and suppose that $L(i_1) = L$ is constant and that $R_1 = R_2 = R$. In this case v can be obtained by solving the following system of two differential equations:

$$\begin{cases} v'(t) = w(t), \\ w'(t) = -\dfrac{1}{LC}\left(\dfrac{L}{R} + RC\right) w(t) - \dfrac{2}{LC} v(t) + \dfrac{e}{LC}, \end{cases} \tag{8.66}$$

with initial conditions $v(0) = 0$, $w(0) = 0$. The system has been obtained from the second-order differential equation

$$LC\frac{d^2v}{dt^2} + \left(\frac{L}{R_2} + R_1 C\right)\frac{dv}{dt} + \left(\frac{R_1}{R_2} + 1\right) v = e. \tag{8.67}$$

We set $L = 0.1$ Henry, $C = 10^{-3}$ Farad, $R = 10$ Ohm and $e = 5$ Volt, where Henry, Farad, Ohm and Volt are respectively the unit measure of inductance, capacitance, resistance and voltage. Now we apply the forward Euler method with $h = 0.001$ seconds in the time interval $[0, 0.1]$, by the Program 8.1:

```
L=0.1; C=1.e-03; R=10; e=5;
[t,u]=feuler(@fsys,[0,0.1],[0 0],100,L,C,R,e);
```

where `fsys` is contained in the file `fsys.m`:

```
function fn=fsys(t,y,L,C,R,e)
LC = L*C;
[n,m]=size(y); fn=zeros(n,m);
fn(1)=y(2);
fn(2)=-(L/R+R*C)/(LC)*y(2)-2/(LC)*y(1)+e/(LC);
return
```

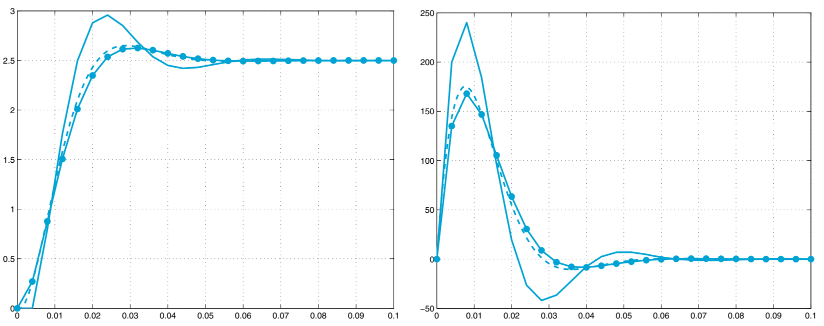


Figure 8.17. Numerical solutions of system (8.66). The potential drop $v(t)$ is reported on the left, its derivative $w(t)$ on the right: the dashed line represents the solution obtained for $h = 0.001$ with the forward Euler method, the solid line is for the one generated via the same method with $h = 0.004$, and the solid line with circles is for the one produced via the Newmark method (8.71) (with $\zeta = 1/4$ and $\theta = 1/2$) with $h = 0.004$

In Figure 8.17 we report the approximated values of $v(t)$ and $w(t)$. As expected, $v(t)$ tends to $e/2 = 2.5$ Volt for $t \to \infty$. In this case, the matrix $\mathrm{A} = [\partial \mathbf{F}/\partial \mathbf{y}](t, \mathbf{y}) = [0, 1; -20000, -200]$, hence does not depend on time. Its eigenvalues are $\lambda_{1,2} = -100 \pm 100i$, so that the bound on time-step which guarantees absolute stability is $h < -2Re(\lambda_i)/|\lambda_i|^2 = 0.01$. ■

Sometimes numerical approximations can be directly derived on the high order equation without passing through the equivalent first order system. Consider for instance the case of the 2nd order Cauchy problem

$$\begin{cases} y''(t) = f(t, y(t), y'(t)) \qquad t \in (t_0, T], \\ y(t_0) = \alpha_0, \quad y'(t_0) = \beta_0. \end{cases} \tag{8.68}$$

Two sequences u_n and v_n will approximate $y(t_n)$ and $y'(t_n)$, respectively. A simple numerical scheme can be constructed as follows: find u_{n+1} such that

$$\frac{u_{n+1} - 2u_n + u_{n-1}}{h^2} = f(t_n, u_n, v_n), \qquad n = 1, \dots, N_h, \tag{8.69}$$

with $u_0 = \alpha_0$ and $v_0 = \beta_0$. Moreover, since $(y_{n+1} - 2y_n + y_{n-1})/h^2$ is a second order approximation of $y''(t_n)$, let us consider a second order approximation for $y'(t_n)$ too, i.e. (see (4.9))

$$v_n = \frac{u_{n+1} - u_{n-1}}{2h}, \text{ with } v_0 = \beta_0. \tag{8.70}$$

The *leap-frog method* (8.69)-(8.70) is accurate of order 2 with respect to h.

A more general method is the *Newmark method*, in which we build two sequences with same meaning as before

$$
\begin{aligned}
u_{n+1} &= u_n + hv_n + h^2\big[\zeta f(t_{n+1}, u_{n+1}, v_{n+1}) \\
&\qquad +(1/2-\zeta)f(t_n, u_n, v_n)\big],
\end{aligned} \tag{8.71}
$$
$$
v_{n+1} = v_n + h\left[(1-\theta)f(t_n, u_n, v_n) + \theta f(t_{n+1}, u_{n+1}, v_{n+1})\right],
$$

with $u_0 = \alpha_0$ and $v_0 = \beta_0$, and ζ and θ are two non-negative real numbers. This method is implicit unless $\zeta = \theta = 0$, second order if $\theta = 1/2$, whereas it is first order accurate if $\theta \neq 1/2$. The condition $\theta \geq 1/2$ is necessary to ensure stability. For $\theta = 1/2$ and $\zeta = 1/4$ we find a rather popular method that is unconditionally stable. However, this method is not suitable for simulations on long time intervals as it introduces oscillatory spurious solutions. For these simulations it is preferable to use $\theta > 1/2$ and $\zeta > (\theta + 1/2)^2/4$ even though the method degenerates to a first order one.

In Program 8.8 we implement the Newmark method. The vector `param` allows to specify the values of the coefficients (`param(1)=`ζ, `param(2)=`θ).

Program 8.8. newmark: Newmark method

```
function [t,u]=newmark(odefun,tspan,y0,Nh,param,...
               varargin)
%NEWMARK Solves second order differential equations
%  using the Newmark method
%  [T,Y]=NEWMARK(ODEFUN,TSPAN,Y0,NH,PARAM) with TSPAN =
%  [T0 TF] integrates the system of differential
%  equations y''=f(t,y,y') from time T0 to TF with
%  initial conditions Y0=(y(t0),y'(t0)) using the
%  Newmark method on an equispaced grid of NH steps.
%  PARAM holds parameters zeta and theta
%  Function ODEFUN(T,Y) must return a vector, whose
%  elements hold the evaluation of f(t,y), of the
%  same dimension of Y.
%  Each row in the solution array Y corresponds to a
%  time returned in the  column vector T.
tt=linspace(tspan(1),tspan(2),Nh+1);
y=y0(:); u=y.';
global glob_h glob_t glob_y glob_odefun;
global glob_zeta glob_theta glob_varargin glob_fn;
glob_h=(tspan(2)-tspan(1))/Nh;
glob_y=y; glob_odefun=odefun;
glob_zeta = param(1); glob_theta = param(2);
glob_varargin=varargin;
if ( exist('OCTAVE_VERSION') )
o_ver=OCTAVE_VERSION;
version=str2num([o_ver(1),o_ver(3),o_ver(5)]);
end

if ( ~exist( 'OCTAVE_VERSION' )  | version >= 320 )
 options=optimset;
 options.Display='off';
 options.TolFun=1.e-12;
 options.MaxFunEvals=10000;
end
glob_fn =odefun(tt(1),glob_y,varargin{:});
for glob_t=tt(2:end)
if ( exist( 'OCTAVE_VERSION' ) & version < 320 )
  w = fsolve('newmarkfun', glob_y );
else
  w = fsolve(@(w) newmarkfun(w),glob_y,options);
end
  glob_fn =odefun(glob_t,w,varargin{:});
  u = [u; w.']; glob_y = w;
end
t=tt';
clear glob_h glob_t glob_y glob_odefun;
clear glob_zeta glob_theta glob_varargin glob_fn;
end

function z=newmarkfun(w)
 global glob_h glob_t glob_y glob_odefun;
 global glob_zeta glob_theta glob_varargin glob_fn;
 fn1=glob_odefun(glob_t,w,glob_varargin{:});
 z(1)=w(1) - glob_y(1) -glob_h*glob_y(2)-...
   glob_h^2*(glob_zeta*fn1+(0.5-glob_zeta)*glob_fn);
 z(2)=w(2) - glob_y(2) -...
   glob_h*((1-glob_theta)*glob_fn+glob_theta*fn1);
end
```

Example 8.9 (Electrical circuits) We consider again the circuit of Problem 8.4 and we solve the second order equation (8.67) with the Newmark scheme. In Figure 8.17 we compare the numerical approximations of the function v computed using the forward Euler scheme (*dashed line* for $h = 0.001$ and *continuous line* for $h = 0.004$) and the Newmark scheme with $\theta = 1/2$ and $\zeta = 1/4$ (*solid line with circles*), with the time-step $h = 0.004$. The better accuracy of the latter solution is due to the fact that the method (8.71) is second order accurate with respect to h. ■

See the Exercises 8.18-8.20.

8.10 Some examples

We end this chapter by considering and solving three non-trivial examples of systems of ordinary differential equations.

8.10.1 The spherical pendulum

The motion of a point $\mathbf{x}(t) = (x_1(t), x_2(t), x_3(t))^T$ with mass m subject to the gravity force $\mathbf{F} = (0, 0, -gm)^T$ (with $g = 9.8$ m/s^2) and constrained to move on the spherical surface of equation $\Phi(\mathbf{x}) =$

$x_1^2 + x_2^2 + x_3^2 - 1 = 0$ is described by the following system of ordinary differential equations

$$\ddot{\mathbf{x}} = \frac{1}{m}\left(\mathbf{F} - \frac{m\,\dot{\mathbf{x}}^T\ \mathrm{H}\,\dot{\mathbf{x}} + \nabla\Phi^T\mathbf{F}}{|\nabla\Phi|^2}\nabla\Phi\right) \text{ for } t > 0. \tag{8.72}$$

We denote by $\dot{\mathbf{x}}$ the first derivative with respect to t, with $\ddot{\mathbf{x}}$ the second derivative, with $\nabla\Phi$ the spatial gradient of Φ, equal to $2\mathbf{x}$, with H the Hessian matrix of Φ whose components are $\mathrm{H}_{ij} = \partial^2\Phi/\partial x_i\partial x_j$ for $i, j = 1, 2, 3$. In our case H is a diagonal matrix with coefficients all equal to 2. System (8.72) must be provided with the initial conditions $\mathbf{x}(0) = \mathbf{x}_0$ and $\dot{\mathbf{x}}\,(0) = \mathbf{v}_0$.

To numerically solve (8.72) let us transform it into a system of differential equations of order 1 in the new variable $\mathbf{y}$, a vector with 6 components. Having set $y_i = x_i$ and $y_{i+3} = \dot{x_i}$ with $i = 1, 2, 3$, and

$$\lambda = \frac{m(y_4, y_5, y_6)^T\mathrm{H}(y_4, y_5, y_6) + \nabla\Phi^T\mathbf{F}}{|\nabla\Phi|^2},$$

we obtain, for $i = 1, 2, 3$,

$$\begin{aligned} \dot{y_i} &= y_{3+i}, \\ \dot{y}_{3+i} &= \frac{1}{m}\left(F_i - \lambda\frac{\partial\Phi}{\partial y_i}\right). \end{aligned} \tag{8.73}$$

We apply the Euler and Crank-Nicolson methods. Initially it is necessary to define a MATLAB *function* (`fvinc` in Program 8.9) which yields the expressions of the right-hand terms (8.73). Furthermore, let us suppose that the initial conditions are given by vector `y0=[0,1,0,.8,0,1.2]` and that the integration interval is `tspan=[0,25]`. We recall the explicit Euler method in the following way

```
[t,y]=feuler(@fvinc,tspan,y0,nt);
```

(the backward Euler `beuler` and Crank-Nicolson `cranknic` methods can be called in the same way), where `nt` is the number of intervals (of constant width) used to discretize the interval `[tspan(1),tspan(2)]`. In the graphs in Figure 8.18 we report the trajectories obtained with 10000 and 100000 discretization nodes. Only in the second case, the solution looks reasonably accurate. As a matter of fact, although we do not know the exact solution to the problem, we can have an idea of the accuracy by noticing that the solution satisfies $r(\mathbf{y}) \equiv |y_1^2 + y_2^2 + y_3^2 - 1| = 0$ and by consequently measuring the maximal value of the residual $r(\mathbf{y}_n)$ when n varies, $\mathbf{y}_n$ being the approximation of the exact solution generated at time t_n. By using 10000 discretization nodes we find $r = 1.0578$, while with 100000 nodes we have $r = 0.1111$, in accordance with the theory requiring the explicit Euler method to converge with order 1.

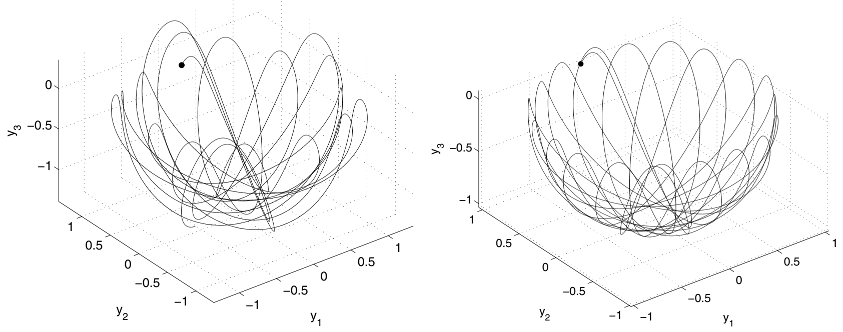


Figure 8.18. The trajectories obtained with the explicit Euler method with $h = 0.0025$ (*on the left*) and $h = 0.00025$ (*on the right*). The blackened point shows the initial datum

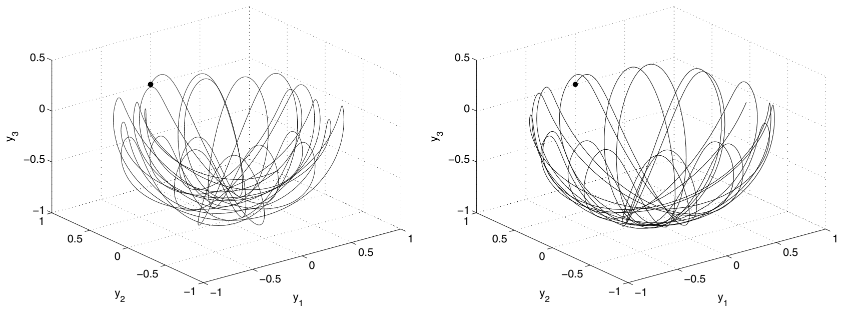


Figure 8.19. The trajectories obtained using the implicit Euler method with $h = 0.00125$ (*on the left*) and using the Crank-Nicolson method with $h = 0.025$ (*on the right*)

By using the implicit Euler method with 20000 steps we obtain the solution reported in Figure 8.19, while the Crank-Nicolson method (of order 2) with only 1000 steps provides the solution reported in the same figure on the right, which is undoubtedly more accurate. Indeed, we find $r = 0.5816$ for the implicit Euler method and $r = 0.0928$ for the Crank-Nicolson method.

As a comparison, let us solve the same problem using the explicit adaptive methods of type Runge-Kutta `ode23` and `ode45`, featured in MATLAB. These (unless differently specified) modify the integration step in order to guarantee that the relative error on the solution is less than 10^{-3} and the absolute error is less than 10^{-6}. We run them using the following commands

```
[t1,y1]=ode23(@fvinc,tspan,y0);
[t2,y2]=ode45(@fvinc,tspan,y0);
```

obtaining the solutions in Figure 8.20.

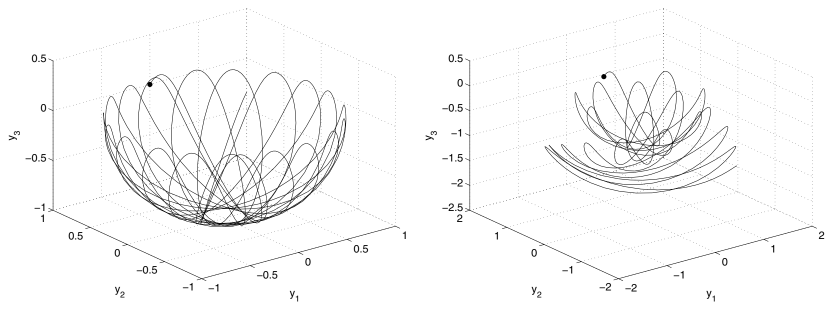


Figure 8.20. The trajectories obtained using methods `ode23` (*left*) and `ode45` (*right*) with the same accuracy criteria. In the second case the error control fails and the solution obtained is less accurate

The two methods used 783, respectively 537, non-uniformly distributed discretization nodes. The residual r is equal to 0.0238 for `ode23` and 3.2563 for `ode45`. Surprisingly, the result obtained with the highest-order method is thus less accurate and this warns us as to using the `ode` programs available in MATLAB. An explanation of this behavior is in the fact that the error estimator implemented in `ode45` is less constraining than that in `ode23`. By slightly decreasing the relative tolerance (it is sufficient to set `options=odeset('RelTol',1.e-04)`) and renaming the program to `[t,y]=ode45(@fvinc,tspan,y0,options);` we can in fact find results comparable with those of `ode23`. Precisely `ode23` requires 1751 discretization nodes and it provides a residual $r = 0.003$, while `ode45` requires 1089 discretization nodes and it provides a residual $r = 0.060$.

Program 8.9. fvinc: forcing term for the spherical pendulum problem

```
function [f]=fvinc(t,y)
[n,m]=size(y); f=zeros(n,m);
phix=2*y(1); phiy=2*y(2); phiz=2*y(3);
H=2*eye(3);
mass=1;
F1=0; F2=0; F3=-mass*9.8;
xp=zeros(3,1);
xp(1:3)=y(4:6);
F=[F1;F2;F3];
G=[phix;phiy;phiz];
lambda=(mass*xp'*H*xp+F'*G)/(G'*G);
f(1:3)=y(4:6);
for k=1:3;
  f(k+3)=(F(k)-lambda*G(k))/mass;
end
return
```

Octave 8.2 `ode23` requires 924 steps while `ode45` requires 575 steps for the same accuracy `tol=1.e-03`.

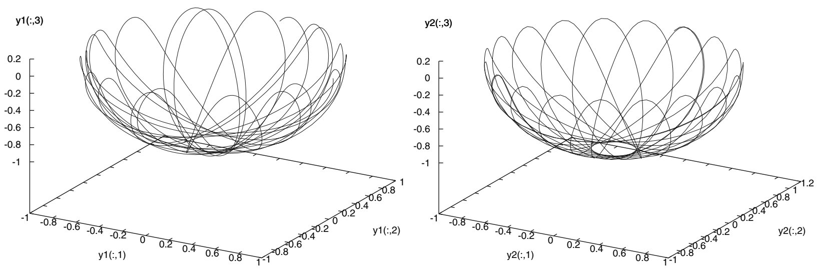


Figure 8.21. The trajectories obtained using methods `ode23` (*left*) and `ode45` (*right*) with the same accuracy criteria.

Note that `ode45` gives results similar to `ode23` as opposed to `ode45` in MATLAB, see Figure 8.21. ■

8.10.2 The three-body problem

We want to compute the evolution of a system composed by three bodies, knowing their initial positions and velocities and their masses under the influence of their reciprocal gravitational attraction. The problem can be formulated by using Newton's laws of motion. However, as opposed to the case of two bodies, there are no known closed form solutions. We suppose that one of the three bodies has considerably larger mass than the two remaining, and in particular we study the case of the Sun-Earth-Mars system, a problem studied by celeber mathematicians such as Lagrange in the eighteenth century, Poincaré towards the end of the nineteenth century and Levi-Civita in the twentieth century.

We denote by M_s the mass of the Sun, by M_e that of the Earth and by M_m that of Mars. The Sun's mass being about 330000 times that of the Earth and the mass of Mars being about one tenth of the Earth's, we can imagine that the center of gravity of the three bodies approximately coincides with the center of the Sun (which will therefore remain still in this model) and that the three objects remain in the plane described by their initial positions. In such case the total force exerted on the Earth will be for instance

$$\mathbf{F}_e = \mathbf{F}_{es} + \mathbf{F}_{em} = M_e \frac{d^2\mathbf{x}_e}{dt^2}, \tag{8.74}$$

where $\mathbf{x}_e = (x_e, y_e)^T$ denotes the Earth's position with respect to the Sun, while $\mathbf{F}_{es}$ and $\mathbf{F}_{em}$ denote the force exerted by the Sun and by Mars, respectively, on the Earth. By applying the universal gravitational law, denoting by G the universal gravity constant and by $\mathbf{x}_m$ the position of Mars with respect to the Sun, equation (8.74) becomes

$$M_e \frac{d^2\mathbf{x}_e}{dt^2} = -GM_eM_s\frac{\mathbf{x}_e}{|\mathbf{x}_e|^3} + GM_eM_m\frac{\mathbf{x}_m - \mathbf{x}_e}{|\mathbf{x}_m - \mathbf{x}_e|^3}.$$

Now, let us take the astronomical unit (1AU) as unit length, the year (1yr) as temporal unit and define the Sun mass as $M_s = \frac{4\pi^2(1\mathrm{AU})^3}{G(1\mathrm{yr})^2}$. By adimensionalizing the previous equations and denoting again with $\mathbf{x}_e$, $\mathbf{x}_m$, $\mathbf{x}_s$ and t the adimensionalized variables, we obtain the following equation

$$\frac{d^2\mathbf{x}_e}{dt^2} = 4\pi^2\left(\frac{M_m}{M_s}\frac{\mathbf{x}_m - \mathbf{x}_e}{|\mathbf{x}_m - \mathbf{x}_e|^3} - \frac{\mathbf{x}_e}{|\mathbf{x}_e|^3}\right). \tag{8.75}$$

A similar equation for planet Mars can be obtained using a similar computation

$$\frac{d^2\mathbf{x}_m}{dt^2} = 4\pi^2\left(\frac{M_e}{M_s}\frac{\mathbf{x}_e - \mathbf{x}_m}{|\mathbf{x}_e - \mathbf{x}_m|^3} - \frac{\mathbf{x}_m}{|\mathbf{x}_m|^3}\right). \tag{8.76}$$

The second-order system (8.75)-(8.76) immediately reduces to a system of eight equations of order one. Program 8.10 allows to evaluate a *function* containing the right-hand side terms of system (8.75)-(8.76).

Program 8.10. threebody: forcing term for the simplified three body system

```
function f=threebody(t,y)
[n,m]=size(y); f=zeros(n,m); Ms=330000; Me=1; Mm=0.1;
D1 = ((y(5)-y(1))^2+(y(7)-y(3))^2)^(3/2);
D2 = (y(1)^2+y(3)^2)^(3/2);
f(1)=y(2); f(2)=4*pi^2*(Me/Ms*(y(5)-y(1))/D1-y(1)/D2);
f(3)=y(4); f(4)=4*pi^2*(Me/Ms*(y(7)-y(3))/D1-y(3)/D2);
D2 = (y(5)^2+y(7)^2)^(3/2);
f(5)=y(6); f(6)=4*pi^2*(Mm/Ms*(y(1)-y(5))/D1-y(5)/D2);
f(7)=y(8); f(8)=4*pi^2*(Mm/Ms*(y(3)-y(7))/D1-y(7)/D2);
return
```

Let us compare the implicit Crank-Nicolson method and the explicit adaptive Runge-Kutta method implemented in `ode23`. Having set the Earth to be 1 unit away from the Sun, Mars will be located at about 1.52 units: the initial position will therefore be $(1, 0)$ for the Earth and $(1.52, 0)$ for Mars. Let us further suppose that the two planets initially have null horizontal velocity and vertical velocity equal to -5.1 units (Earth) and -4.6 units (Mars): this way they should move along reasonably stable orbits around the Sun. For the Crank-Nicolson method we choose 2000 discretization steps:

```
[t23,u23]=ode23(@threebody,[0 10],...
                [1.52 0 0 -4.6 1 0 0 -5.1]);
[tcn,ucn]=cranknic(@threebody,[0 10],...
                   [1.52 0 0 -4.6 1 0 0 -5.1],2000);
```

The graphs in Figure 8.22 show that the two methods are both able to reproduce the elliptical orbits of the two planets around the Sun. Method

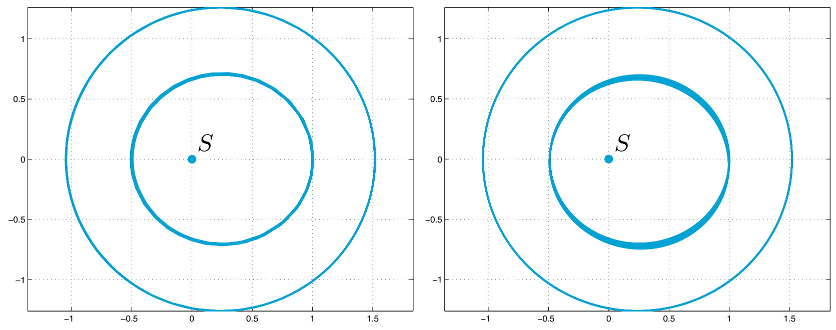


Figure 8.22. The Earth's (*inmost*) and Mars's orbit with respect to the Sun as computed with the adaptive method `ode23` (*on the left*) (with 543 steps) and with the Crank-Nicolson method (*on the right*) (with 2000 steps)

`ode23` only required 543 (nonuniform) steps to generate a more accurate solution than that generated by an implicit method with the same order of accuracy, but which does not use step adaptivity.

Octave 8.3 `ode23` requires 847 steps to generate a solution with a tolerance of 1e-3. ■

8.10.3 Some stiff problems

Let us consider the following differential problem, proposed by [Gea71], as a variant of the model problem (8.28):

$$\begin{cases} y'(t) = \lambda(y(t) - g(t)) + g'(t), & t > 0, \\ y(0) = y_0, \end{cases} \tag{8.77}$$

where g is a regular function and $\lambda < 0$ has a very large absolute value, whose solution

$$y(t) = (y_0 - g(0))e^{\lambda t} + g(t), \qquad t \geq 0. \tag{8.78}$$

is the sum of two components, also called *transient* and *persistent* solution, respectively. Initially, on a time interval of length $\mathcal{O}(1/\lambda)$, the transient component prevails, whereas the persistent component becomes predominant in the asymptotic regime (for sufficiently large t).

In particular, we set $g(t) = t$, $\lambda = -100$, and $y_0 = 1$ and solve problem (8.77) over the interval $(0, 100)$ using the explicit Euler method: since in this case $f(t, y) = \lambda(y(t) - g(t)) + g'(t)$ we have $\partial f/\partial y = \lambda$, and the stability analysis performed in Section 8.5 suggests that we choose $h < 2/100$. This restriction is dictated by the presence of the component behaving like e^{-100t} and appears completely unjustified when we think

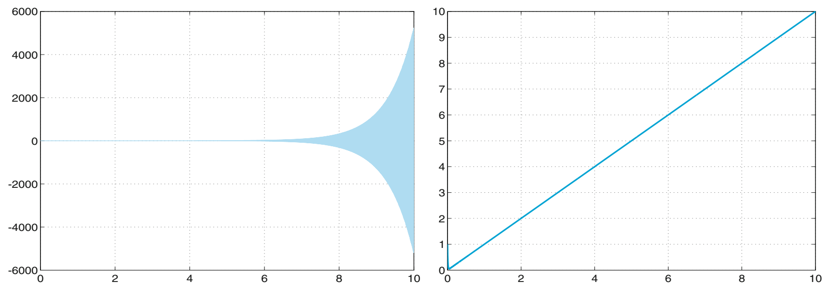


Figure 8.23. Solutions obtained using method (8.59) for problem (8.77) violating the stability condition ($h = 0.0055$, *left*) and respecting it ($h = 0.0054$, *right*)

of its weight with respect to the whole solution for sufficiently large t (to get an idea, for $t = 1$ we have $e^{-100} \approx 10^{-44}$). The situation gets worse using a higher order explicit method, such as for instance the Adams-Bashforth (8.59) method of order 3: the absolute stability region reduces (see Figure 8.13) and, consequently, the restriction on h becomes even stricter, $h < 0.00545$. Violating – even slightly – such restriction produces unacceptable solutions (as shown in Figure 8.23 on the left).

We thus face an apparently simple problem, but one that becomes difficult to solve with an explicit method (and more generally with a method which is not A-stable).

In fact, even though for large values of t it is the persistent component of the solution that prevails (in the current case it is a straight line), yet for its correct approximation we must enforce a strong limitation on h. Such kind of problem is called *stiff*, or, more precisely, it is a stiff problem on the interval on which the persistent solution prevails. As a matter of fact the choice of h is subjected to stability constraints; in these cases, the use of explicit methods, even if implemented using adaptive strategies, is prohibitive.

Programs implementing adaptive methods do not explicitely check that absolute stability condition is satisfied. Nevertheless, the error estimator provides steplength h such that $h\lambda$ belongs to the absolute stability region.

We consider now a system of linear differential equations that reads

$$\mathbf{y}'(t) = \mathrm{A}\mathbf{y}(t) + \boldsymbol{\varphi}(t), \qquad \mathrm{A} \in \mathbb{R}^{n\times n}, \quad \boldsymbol{\varphi}(t) \in \mathbb{R}^n, \tag{8.79}$$

where A has n distinct eigenvalues λ_j, $j = 1, \dots, n$ with $\mathrm{Re}(\lambda_j) < 0$. Its exact solution is

$$\mathbf{y}(t) = \sum_{j=1}^{n} C_j e^{\lambda_j t}\mathbf{v}_j + \boldsymbol{\psi}(t), \tag{8.80}$$

where $C_1, \dots, C_n$ are n costants and $\{\mathbf{v}_j\}$ is a basis of $\mathbb{R}^n$ whose components are the eigenvectors of A, while $\boldsymbol{\psi}(t)$ is a special solution of (8.79).

Similarly to the scalar case (8.78), $C_j e^{\lambda_j t}\mathbf{v}_j$ represent the transient components of the solution and $\boldsymbol{\psi}(t)$ the persistent component (for large t). If $|\mathrm{Re}(\lambda_j)|$ is large, the corresponding transient component will tend to zero very quickly, while for small values of $|\mathrm{Re}(\lambda_j)|$, the corresponding transient components will decay more slowly. If we approximate (8.79) by a numerical scheme that is not absolutely stable, the transient component featuring the largest value of $|\mathrm{Re}(\lambda_j)|$ is the one that yields the most stringent constraint on the steplength h, even though such component is the quickest to decay to zero.

A parameter that is often used to measure the stiff character of a system is

$$r_s = \frac{\max_j |\mathrm{Re}(\lambda_j)|}{\min_j |\mathrm{Re}(\lambda_j)|},$$

even though by itself r_s is not fully meaningful. As a matter of fact, the stiff character of a system depends on r_s, the eigenvalues of A, the initial conditions, the persistent component of the solution and the time interval on which the system has to be solved.

On the other hand, the stiff character depends not only on the form of the exact solution of (8.79); as a matter of fact there exist different systems, some of them stiff, some other non-stiff, all featuring the same exact solution, see, e.g., [Lam91, Ch. 6].
How can we therefore state whether a system is stiff or not? Let us quote the following definition proposed by [Lam91, pag. 220].

Definition 8.1 *A system of ordinary differential equations is said* stiff *if, once approximated by a numerical method featuring an absolute stability region of bounded extension, "forces" the said numerical method, for every initial condition for which the given problem admits a solution, to use a steplength exceedingly small with respect to the one that would be necessary to reasonably reproduce the behavior of the exact solution.*

In the case of problem (8.77) (or (8.79)) the system is not stiff in the initial interval where the solution varies quickly, whence the need of adopting a small h to well capture the sharp layer. Rather, it is stiff in the next interval where the solution features a mild slope. Within this interval the fastest transient, although exhausted because negligible with respect to the other components, still dictates the choice of a tiny steplength h because of stability constraints.

A-stable numerical methods (those whose absolute stability region comprises the half complex plane $Re\lambda < 0$) with adaptive choice of the steplength are the most efficient for *stiff* problems. Their implicit character makes them more computationally involved than explicit methods, however they can afford much larger steplengths. Explicit methods, on their turn, may be unaffordable because of the strong limitation on h.

The algorithm implemented in function `ode15s` is based on multistep methods and backward differentiation formulas BDF introduced in Section 8.7. Its formal convergence order is variable and at most 5. This method is very effective also on systems that are non-stiff for which the Jacobian matrix of $\mathbf{f}(t, \mathbf{y})$ is either constant or features very small variations.

`ode23s` The function `ode23s` implements a linear implicit multistep method based on Rosenbrock methods see [SR97] for a detailed description of these two functions.

Example 8.10 Let us consider the system $\mathbf{y}'(t) = \mathrm{A}\mathbf{y}(t)$, $t \in (0, 100)$ with initial condition $\mathbf{y}(0) = \mathbf{y}_0$, where $\mathbf{y} = (y_1, y_2)^T$, $\mathbf{y}_0 = (y_{1,0}, y_{2,0})^T$ and

$$\mathrm{A} = \begin{bmatrix} 0 & 1 \\ -\lambda_1\lambda_2 & \lambda_1 + \lambda_2 \end{bmatrix},$$

where λ_1 and λ_2 are two different negative numbers such that $|\lambda_1| \gg |\lambda_2|$. Matrix A has eigenvalues λ_1 and λ_2 and eigenvectors $\mathbf{v}_1 = (1, \lambda_1)^T$, $\mathbf{v}_2 = (1, \lambda_2)^T$. Thanks to (8.80) the exact solution is

$$\mathbf{y}(t) = \begin{pmatrix} C_1 e^{\lambda_1 t} + C_2 e^{\lambda_2 t} \\ C_1\lambda_1 e^{\lambda_1 t} + C_2\lambda_2 e^{\lambda_2 t} \end{pmatrix}^T. \tag{8.81}$$

The constants C_1 and C_2 are obtained by fulfilling the initial condition:

$$C_1 = \frac{\lambda_2 y_{1,0} - y_{2,0}}{\lambda_2 - \lambda_1}, \qquad C_2 = \frac{y_{2,0} - \lambda_1 y_{1,0}}{\lambda_2 - \lambda_1}.$$

Based on the remarks made earlier, the integration step of an explicit method used for the resolution of such a system will depend uniquely on the eigenvalue having largest modulus, λ_1. Let us assess this experimentally using the explicit Euler method and choosing $\lambda_1 = -100$, $\lambda_2 = -1$ (therefore $r_s = 100$), $y_{1,0} = y_{2,0} = 1$. In Figure 8.24 we report the solutions computed by violating (*left*) or respecting (*right*) the stability condition $h < 1/50$. ■

The definition of stiff problem can be extended, with some care, to the nonlinear case (see for instance [QSS07, Chapter 11]). One of the most studied nonlinear *stiff* problems is given by the *Van der Pol equation*

$$\frac{d^2x}{dt^2} = \mu(1 - x^2)\frac{dx}{dt} - x, \tag{8.82}$$

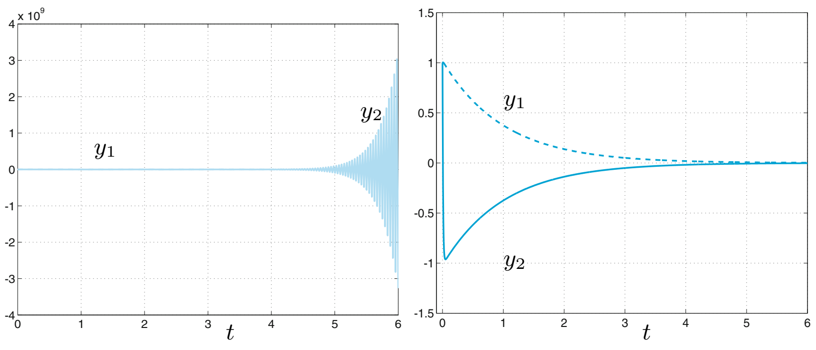


Figure 8.24. Solutions to the problem in Example 8.10 for $h = 0.0207$ (*left*) and $h = 0.01$ (*right*). In the first case the condition $h < 2/|\lambda_1| = 0.02$ is violated and the method is unstable. The second case features a strong variation of the fast transient component y_2. Consider the totally different scale in the two graphs

proposed in 1920 and used in the study of circuits containing thermionic valves, the so-called vacuum tubes, such as cathodic tubes in television sets or magnetrons in microwave ovens.

If we set $\mathbf{y} = (x, z)^T$, with $z = dx/dt$, (8.82) is equivalent to the following nonlinear first order system

$$\mathbf{y}' = \mathbf{F}(t, \mathbf{y}) = \begin{bmatrix} z \\ -x + \mu(1 - x^2)z \end{bmatrix}. \tag{8.83}$$

Such system becomes increasingly stiff with the increase of the μ parameter. In the solution we find in fact two components which denote totally different dynamics with the increase of μ. The one having the fastest dynamics imposes a limitation on the integration step which gets more and more prohibitive with the increase of μ.

If we solve (8.82) using `ode23` and `ode45`, we realize that these are too costly when μ is large. With $\mu = 100$ and initial condition $\mathbf{y} = (1, 1)^T$, `ode23` requires 7835 steps and `ode45` 23473 steps to integrate between $t = 0$ and $t = 100$. Reading the MATLAB *help* we discover that these methods are based on explicit schemes and therefore they are not recommended for stiff problems: for these, other procedures are suggested, such as for instance the implicit methods `ode23s` or `ode15s`. The difference in terms of number of steps is remarkable, as shown in Table 8.1. Notice however that the number of steps for `ode23s` is smaller than that for `ode23` only for large enough values of μ (thus for very stiff problems).

`ode23s`

Example 8.11 (Chemical kinetics) We want to investigate the temporal behavior of chemical reactions of species in homogeneous media. Quite often,

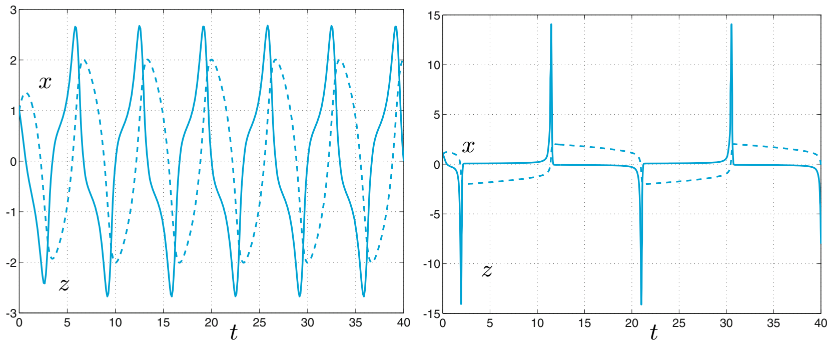


Figure 8.25. Behavior of the components of the solutions $\mathbf{y}$ to system (8.83) for $\mu = 1$ (*left*) and $\mu = 10$ (*right*)

Table 8.1. Behavior of the number of integration steps for various approximation methods with growing μ parameter

μ	`ode23`	`ode45`	`ode23s`	`ode15s`
0.1	471	509	614	586
1	775	1065	838	975
10	1220	2809	1005	1077
100	7835	23473	299	305
1000	112823	342265	183	220

both fast and slow species cohexist, that evolve according to differente characteristic times. Below we consider a mathematical model that represents a simplified version of this process. This model, named Davis-Skodje (see, e.g., [VGCN05]), addresses two species $y_1(t)$ and $y_2(t)$ that evolve according to the equations

$$\begin{cases} \dfrac{dy_1}{dt} = \dfrac{1}{\varepsilon}\left(-y_1 + \dfrac{y_2}{1+y_2}\right) - \dfrac{y_2}{(1+y_2)^2}, & t > 0 \\ \dfrac{dy_2}{dt} = -y_2, & t > 0 \\ y_1(0) = y_{1,0} \\ y_2(0) = y_{2,0}, \end{cases} \tag{8.84}$$

where $\varepsilon > 0$, $y_{1,0}$ and $y_{2,0}$ are given.

The exact solution is:

$$y_1(t) = \left(y_{1,0} - \frac{y_{2,0}}{1+y_{2,0}}\right)e^{-t/\varepsilon} + \frac{y_{2,0}e^{-t}}{1+y_{2,0}e^{-t}}$$
$$y_2(t) = y_{2,0}e^{-t}.$$

The ratio $1/\varepsilon$ is a measure of the system's stiffness: the larger $1/\varepsilon$ the wider the gap between the temporal scales of the evolution of the two species, than the more complex is the numerical computation.

To numerically solve system (8.84) with $\varepsilon = 10^{-6}$ and initial condition $\mathbf{y}_0 = (1.5, 1)^T$, we have defined the function

Table 8.2. Number of steps used by a few MATLAB functions to solve problem (8.84) for different values of the parameter ε

ε	`ode23`	`ode45`	`ode23s`	`ode15s`
10^{-2}	409	1241	73	73
10^{-3}	3991	12081	84	81
10^{-4}	39808	120553	87	85

```
function [f]=funds(t,y)
epsilon=1.e-6; [n,m]=size(y);f=zeros(n,m);
f(1)=-1/epsilon*y(1)+( (1/epsilon-1)*y(2)+...
      1/epsilon*y(2)*y(2))/(1+y(2))^2;
f(2)=-y(2);
end
```

Then we call the MATLAB function `ode23s` by the following commands

```
y0=[1.5,1]; tspan=[0,10];
[t,y]=ode23s(@funds,tspan,y0);
```

In Table 8.2 we report the number of steps required by the explicit methods `ode23`, `ode45`, and by the implicit methods `ode24s`, `ode15s`. We can appreciate the better efficiency of methods `ode23s` and `ode15s`, as they have been specifically designed for stiff equations.

In Figure 8.26, left, we plot numerical solutions: the species y_1 evolves very quickly at the beginning of the simulation during a time interval of length $\mathcal{O}(\varepsilon)$, and very slowly after. On the contrary, the species y_2 varies slowly and uniformly during the whole simulation time.

In Figure 8.26, right, trajectories of problem (8.84) are shown for $\varepsilon = 10^{-6}$ and with several initial conditions $[y_{1,0}, y_{2,0}]^T$. Horizontal stretches of trajectories are covered in a very short initial time interval of length $\mathcal{O}(\varepsilon)$, while the curved ways are covered in the remaining time of length $10 - \mathcal{O}(\varepsilon)$. Analysis of trajectories can be useful to acquire carachteristic information of the chemical process. ■

Octave 8.4 While `ode15s` and `ode23s` are not available in Octave, many ODE solvers capable of dealing with stiff problems are available in the Octave core (`lsode`, `dassl`, `daspk`) and in the `odepkg` package from Octave-Forge (`ode2r`, `ode5r`, `odebda`, `oders`, `odesx`). ■

8.11 What we haven't told you

For a complete derivation of the whole family of the Runge-Kutta methods we refer to [But87], [Lam91] and [QSS07, Chapter 11].

For derivation and analysis of multistep methods we refer to [Arn73] and [Lam91].

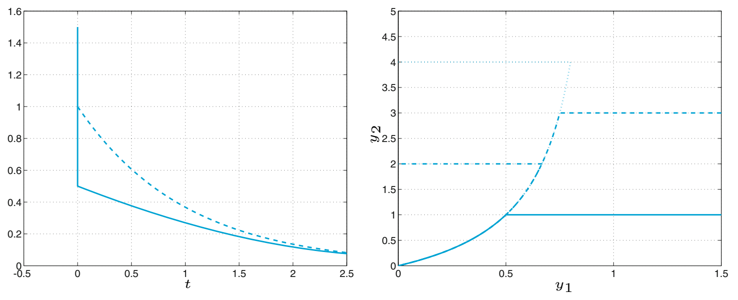


Figure 8.26. At left, numerical solutions ($y_1(t)$ *(continuous line)* and $y_2(t)$ *(dashed line)*) of system (8.84) with initial conditions $y_{1,0} = 1.5$, $y_{2,0} = 1$. At right, trajectories of (8.84) for several initial data $\mathbf{y}_0 = (y_{1,0}, y_{2,0})^T$: $\mathbf{y}_0 = (1.5, 1)^T$ *(continuous line)*, $(1.5, 3)^T$ *(dashed line)*, $(0, 2)^T$ *(dotted-dashed line)*, $(0, 4)^T$ *(dotted line)*. $\varepsilon = 10^{-6}$ in all simulations

8.12 Exercises

Exercise 8.1 Apply the backward Euler and forward Euler methods for the solution of the Cauchy problem

$$y' = \sin(t) + y, \; t \in (0, 1], \text{ with } y(0) = 0, \tag{8.85}$$

and verify that both converge with order 1.

Exercise 8.2 Consider the Cauchy problem

$$y' = -te^{-y}, \; t \in (0, 1], \text{ with } y(0) = 0. \tag{8.86}$$

Apply the forward Euler method with $h = 1/100$ and estimate the number of exact significant digits of the approximate solution at $t = 1$ (use the property that the value of the exact solution is included between -1 and 0).

Exercise 8.3 The backward Euler method applied to problem (8.86) requires at each step the solution of the nonlinear equation: $u_{n+1} = u_n - ht_{n+1}e^{-u_{n+1}} = \phi(u_{n+1})$. The solution u_{n+1} can be obtained by the following fixed-point iteration:
for $k = 0, 1, \ldots$, compute $u_{n+1}^{(k+1)} = \phi(u_{n+1}^{(k)})$, with $u_{n+1}^{(0)} = u_n$.
Find under which restriction on h these iterations converge.

Exercise 8.4 Repeat Exercise 8.1 for the Crank-Nicolson method.

Exercise 8.5 Verify that the Crank-Nicolson method can be derived from the following integral form of the Cauchy problem (8.5)

$$y(t) - y_0 = \int_{t_0}^{t} f(\tau, y(\tau))d\tau$$

provided that the integral is approximated by the trapezoidal formula (4.19).

Exercise 8.6 Solve the model problem (8.28) with $\lambda = -1+i$ by the forward Euler method and find the values of h for which we have absolute stability.

Exercise 8.7 Show that the Heun method defined in (8.64) is consistent. Write a MATLAB program to implement it for the solution of the Cauchy problem (8.85) and verify experimentally that the method has order of convergence equal to 2 with respect to h.

Exercise 8.8 Prove that the Heun method (8.64) is absolutely stable if $-2 < h\lambda < 0$ where λ is real and negative.

Exercise 8.9 Prove formula (8.34).

Exercise 8.10 Prove the inequality (8.39).

Exercise 8.11 Prove the inequality (8.40).

Exercise 8.12 Verify the consistency of the RK3 method (8.58). Write a MATLAB program to implement it for the solution of the Cauchy problem (8.85) and verify experimentally that the method has order of convergence equal to 3 with respect to h. The methods (8.64) and (8.58) stand at the base of the MATLAB program `ode23` for the solution of ordinary differential equations.

Exercise 8.13 Prove that the method (8.58) is absolutely stable if $-2.5 < h\lambda < 0$ where λ is real and negative.

Exercise 8.14 The *modified Euler method* is defined as follows:

$$u^*_{n+1} = u_n + hf(t_n, u_n),\ u_{n+1} = u_n + hf(t_{n+1}, u^*_{n+1}). \qquad (8.87)$$

Find under which condition on h this method is absolutely stable.

Exercise 8.15 (Thermodynamics) Solve equation (8.1) by the Crank-Nicolson method and the Heun method when the body in question is a cube with side equal to 1 m and mass equal to 1 Kg. Assume that $T_0 = 180K$, $T_e = 200K$, $\gamma = 0.5$ and $C = 100J/(Kg/K)$. Compare the results obtained by using $h = 20$ and $h = 10$, for t ranging from 0 to 200 seconds.

Exercise 8.16 Use MATLAB to compute the region of absolute stability of the Heun method.

Exercise 8.17 Solve the Cauchy problem (8.16) by the Heun method and verify its order.

Exercise 8.18 The displacement $x(t)$ of a vibrating system represented by a body of a given weight and a spring, subjected to a resistive force proportional to the velocity, is described by the second-order differential equation $x''+5x'+6x = 0$. Solve it by the Heun method assuming that $x(0) = 1$ and $x'(0) = 0$, for $t \in [0, 5]$.

Exercise 8.19 The motion of a frictionless Foucault pendulum is described by the system of two equations

$$x'' - 2\omega\sin(\Psi)y' + k^2x = 0, \quad y'' + 2\omega\cos(\Psi)x' + k^2y = 0,$$

where Ψ is the latitude of the place where the pendulum is located, $\omega = 7.29 \cdot 10^{-5}$ sec^{-1} is the angular velocity of the Earth, $k = \sqrt{g/l}$ with $g = 9.8$ m/sec^2 and l is the length of the pendulum. Apply the forward Euler method to compute $x = x(t)$ and $y = y(t)$ for t ranging between 0 and 300 seconds and $\Psi = \pi/4$.

Exercise 8.20 (Baseball trajectory) Using `ode23`, solve Problem 8.3 by assuming that the initial velocity of the ball be $\mathbf{v}(0) = v_0(\cos(\phi), 0, \sin(\phi))^T$, with $v_0 = 38$ m/s, $\phi = 1$ degree and an angular velocity equal to $180 \cdot 1.047198$ radiants per second. If $\mathbf{x}(0) = \mathbf{0}$, after how many seconds (approximately) will the ball touch the ground (i.e., $z = 0$)?

Exercise 8.21 (Chemical kinetics) Given the real values $y_{1,0}$, $y_{2,0}$ e $y_{3,0}$, the following system

$$\begin{cases} \dfrac{dy_1}{dt} = \dfrac{1}{\varepsilon}(-5y_1 - y_1y_2 + 5y_2^2 + y_3) + y_2y_3 - y_1, & t > 0 \\ \dfrac{dy_2}{dt} = \dfrac{1}{\varepsilon}(10y_1 - y_1y_2 - 10y_2^2 + y_3) - y_2y_3 + y_1, & t > 0 \\ \dfrac{dy_3}{dt} = \dfrac{1}{\varepsilon}(y_1y_2 - y_3) - y_2y_3 + y_1, & t > 0 \\ y_1(0) = y_{1,0} \\ y_2(0) = y_{2,0}, y_3(0) = y_{3,0}, \end{cases} \tag{8.88}$$

simulates the evolution of the concentration of three species in a chemical reaction. By fixing the initial datum $\mathbf{y}_0 = (1, 0.5, 0)^T$ and setting $\varepsilon = 10^{-2}$, solve system (8.88) with $t \in [0, 10]$, calling `ode23` and `ode23s`, then comment on the stiffness of the system. Finally, plot the computed solution in the phase space, for different values of the initial datum $\mathbf{y}_0 = (y_{1,0}, y_{2,0}, y_{3,0})^T$ with $0 \leq y_{i,0} \leq 1$ and $i = 1, 3$.

9

Numerical approximation of boundary-value problems

Boundary-value problems are differential problems set in an interval (a, b) of the real line or in an open multidimensional region $\Omega \subset \mathbb{R}^d$ $(d = 2, 3)$ for which the value of the unknown solution (or its derivatives) is prescribed at the end-points a and b of the interval, or on the boundary $\partial\Omega$ of the multidimensional region.

In the multidimensional case the differential equation will involve *partial derivatives* of the exact solution with respect to the space coordinates. Equations depending also on time (denoted with t), like the heat equation and the wave equation, are called initial-boundary-value problems. In that case initial conditions at $t = 0$ need to be prescribed as well.

Some examples of boundary-value problems are reported below.

1. *Poisson equation*:

$$-u''(x) = f(x),\ x \in (a, b), \tag{9.1}$$

or (in several dimensions)

$$-\Delta u(\mathbf{x}) = f(\mathbf{x}),\ \mathbf{x} = (x_1, \ldots, x_d)^T \in \Omega, \tag{9.2}$$

where f is a given function and Δ is the so-called *Laplace operator*:

$$\Delta u = \sum_{i=1}^{d} \frac{\partial^2 u}{\partial x_i^2}.$$

The symbol $\partial \cdot / \partial x_i$ denotes partial derivative with respect to the x_i variable, that is, for every point $\mathbf{x}^0$

$$\frac{\partial u}{\partial x_i}(\mathbf{x}^0) = \lim_{h \to 0} \frac{u(\mathbf{x}^0 + h\mathbf{e}_i) - u(\mathbf{x}^0)}{h}, \tag{9.3}$$

where $\mathbf{e}_i$ is ith unitary vector of $\mathbb{R}^d$.

2. *Heat equation*:

$$\frac{\partial u(x,t)}{\partial t} - \mu \frac{\partial^2 u(x,t)}{\partial x^2} = f(x,t),\ x \in (a,b),\ \ t > 0, \tag{9.4}$$

   or (in several dimensions)

$$\frac{\partial u(\mathbf{x},t)}{\partial t} - \mu \Delta u(\mathbf{x},t) = f(\mathbf{x},t),\ \mathbf{x} \in \Omega,\ \ t > 0, \tag{9.5}$$

   where $\mu > 0$ is a given coefficient representing the thermal diffusivity, and f is again a given function.
3. *Wave equation*:

$$\frac{\partial^2 u(x,t)}{\partial t^2} - c \frac{\partial^2 u(x,t)}{\partial x^2} = 0,\ x \in (a,b),\ \ t > 0,$$

   or (in several dimensions)

$$\frac{\partial^2 u(\mathbf{x},t)}{\partial t^2} - c \Delta u(\mathbf{x},t) = 0,\ \mathbf{x} \in \Omega,\ \ t > 0,$$

   where c is a given positive constant.

For a more complete description of general partial differential equations, the reader is referred for instance to [Eva98], [Sal08], and for their numerical approximation to [Qua13], [QV94], [EEHJ96] or [Lan03].

9.1 Some representative problems

Problem 9.1 (Hydrogeology) The study of filtration in groundwater can lead, in some cases, to an equation like (9.2). Consider a portion Ω occupied by a porous medium (like ground or clay). According to the Darcy law, the water velocity filtration $\mathbf{q} = (q_1, q_2, q_3)^T$ is equal to the variation of the water level ϕ in the medium, precisely

$$\mathbf{q} = -K \nabla \phi, \tag{9.6}$$

where K is the constant hydraulic conductivity of the porous medium and $\nabla \phi$ denotes the spatial gradient of ϕ. Assume that the fluid density is constant; then the mass conservation principle yields the equation $\mathrm{div}\mathbf{q} = 0$, where $\mathrm{div}\mathbf{q}$ is the *divergence* of the vector $\mathbf{q}$ and is defined as

$$\mathrm{div}\mathbf{q} = \sum_{i=1}^{3} \frac{\partial q_i}{\partial x_i}.$$

Thanks to (9.6) we therefore find that ϕ satisfies the Poisson problem $\Delta \phi = 0$ (see Exercise 9.8). ■

Problem 9.2 (Thermodynamics) Let $\Omega \subset \mathbb{R}^d$ be a volume occupied by a continuous medium. Denoting by $\mathbf{J}(\mathbf{x}, t)$ and $T(\mathbf{x}, t)$ the heat flux and the temperature of the medium, respectively, the Fourier law states that heat flux is proportional to the variation of the temperature T, that is

$$\mathbf{J}(\mathbf{x}, t) = -k\nabla T(\mathbf{x}, t),$$

where k is a positive constant expressing the thermal conductivity coefficient. Imposing the conservation of energy, that is, the rate of change of energy of a volume equals the rate at which heat flows into it, we obtain the heat equation

$$\rho c \frac{\partial T}{\partial t} = k\Delta T, \tag{9.7}$$

where ρ is the mass density of the continuous medium and c is the specific heat capacity (per unit mass). If, in addition, heat is produced at the rate $f(\mathbf{x}, t)$ by some other means (e.g., electrical heating), (9.7) becomes

$$\rho c \frac{\partial T}{\partial t} = k\Delta T + f. \tag{9.8}$$

The coefficient $\mu = k/(\rho c)$ is the so-called *thermal diffusivity*. For the solution of this problem see Example 9.4. ■

Problem 9.3 (Communications) We consider a telegraph wire with resistance R and self-inductance L per unit length. Assuming that the current can drain away to ground through a capacitance C and a conductance G per unith length (see Figure 9.1), the equation for the voltage v is

$$\frac{\partial^2 v}{\partial t^2} - c\frac{\partial^2 v}{\partial x^2} = -\alpha \frac{\partial v}{\partial t} - \beta v, \tag{9.9}$$

where $c = 1/(LC)$, $\alpha = R/L + G/C$ and $\beta = RG/(LC)$. Equation (9.9) is an example of a second order hyperbolic equation and it is known as *telegrapher's equation* (or just telegraph equation) (see [Str07]). The solution of this problem is given in Example 9.8. ■

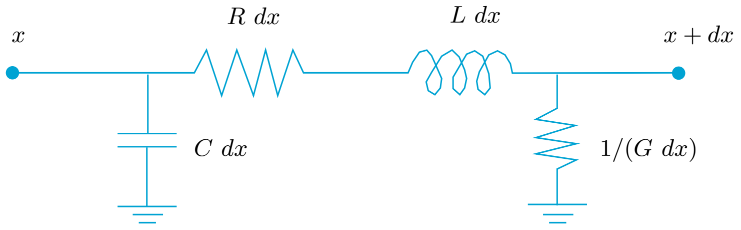


Figure 9.1. An element of cable of length dx

9.2 Approximation of boundary-value problems

The differential equations presented so far feature an infinite number of solutions. With the aim of obtaining a unique solution we must impose suitable conditions on the boundary $\partial\Omega$ of Ω and, for the time-dependent equations, suitable initial conditions at time $t = 0$.

In this section we consider the Poisson equations (9.1) or (9.2). In the one-dimensional case (9.1), to fix the solution one possibility is to prescribe the value of u at $x = a$ and $x = b$, obtaining

$$\boxed{\begin{array}{ll} -u''(x) = f(x) & \text{for } x \in (a, b), \\ u(a) = \alpha, & u(b) = \beta \end{array}} \tag{9.10}$$

where α and β are two given real numbers. This is a *Dirichlet boundary-value problem*, and is precisely the problem that we will face in the next section.
Performing double integration it is easily seen that if $f \in C^0([a, b])$, the solution u exists and is unique; moreover it belongs to $C^2([a, b])$.

Although (9.10) is an ordinary differential problem, it cannot be cast in the form of a Cauchy problem for ordinary differential equations since the value of u is prescribed at two different points.

Instead to set Dirichlet boundary conditions $(9.10)_2$ we can impose $u'(a) = \gamma$, $u'(b) = \delta$ (where γ and δ are suitable constants such that $\gamma - \delta = \int_a^b f(x)\mathrm{d}x$). A problem with these boundary conditions is named *Neumann* problem. Note that its solution is known up to an additive constant.

In the two-dimensional case, the Dirichlet boundary-value problem takes the following form: being given two functions $f = f(\mathbf{x})$ and $g = g(\mathbf{x})$, find a function $u = u(\mathbf{x})$ such that

$$\boxed{\begin{array}{ll} -\Delta u(\mathbf{x}) = f(\mathbf{x}) & \text{for } \mathbf{x} \in \Omega, \\ u(\mathbf{x}) = g(\mathbf{x}) & \text{for } \mathbf{x} \in \partial\Omega \end{array}} \tag{9.11}$$

Alternatively to the boundary condition on (9.11), we can prescribe a value for the partial derivative of u with respect to the normal direction to the boundary $\partial\Omega$, that is

$$\frac{\partial u}{\partial \mathbf{n}}(\mathbf{x}) = \nabla u(\mathbf{x}) \cdot \mathbf{n}(\mathbf{x}) = h(\mathbf{x}) \qquad \text{for } \mathbf{x} \in \partial\Omega,$$

where h is a suitable function such that $\displaystyle\int_{\partial\Omega} h = -\int_\Omega f$ (see Figure 9.2), in which case we will get a *Neumann boundary-value problem.*

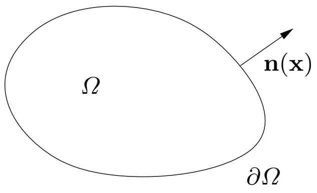


Figure 9.2. A two-dimensional domain Ω and the unit outward normal versor to $\partial\Omega$

It can be proven that if f and g are two continuous functions and the boundary $\partial\Omega$ of the region Ω is regular enough, then the Dirichlet boundary-value problem (9.11) has a unique solution (while the solution of the Neumann boundary-value problem is unique up to an additive constant).

The numerical methods which are used for its solution are based on the same principles used for the approximation of the one-dimensional boundary-value problem. This is the reason why in Sections 9.2.1 and 9.2.3 we will make a digression on the numerical solution of problem (9.10) with either finite difference and finite element methods, respectively.

With this aim we introduce on $[a, b]$ a partition into intervals $I_j = [x_j, x_{j+1}]$ for $j = 0, \ldots, N$ with $x_0 = a$ and $x_{N+1} = b$. We assume for simplicity that all intervals have the same length $h = (b - a)/(N + 1)$.

9.2.1 Finite difference approximation of the one-dimensional Poisson problem

The differential equation (9.10) must be satisfied in particular at any point x_j (which we call *nodes* from now on) internal to (a, b), that is

$$-u''(x_j) = f(x_j), \qquad j = 1, \ldots, N.$$

We can approximate this set of N equations by replacing the second derivative with a suitable finite difference as we have done in Chapter 4 for the first derivatives. In particular, we observe that if $u : [a, b] \to \mathbb{R}$ is a sufficiently smooth function in a neighborhood of a generic point $\bar{x} \in (a, b)$, then the quantity

$$\delta^2 u(\bar{x}) = \frac{u(\bar{x} + h) - 2u(\bar{x}) + u(\bar{x} - h)}{h^2} \tag{9.12}$$

provides an approximation to $u''(\bar{x})$ of order 2 with respect to h (see Exercise 9.3). This suggests the use of the following approximation to problem (9.10): find $\{u_j\}_{j=1}^N$ such that

$$\boxed{-\frac{u_{j+1}-2u_j+u_{j-1}}{h^2}=f(x_j), \qquad j=1,\dots,N} \tag{9.13}$$

with $u_0=\alpha$ and $u_{N+1}=\beta$. Obviously, u_j will be an approximation of $u(x_j)$. Equations (9.13) provide a linear system

$$\mathrm{A}\mathbf{u}_h=h^2\mathbf{f}, \tag{9.14}$$

where $\mathbf{u}_h=(u_1,\dots,u_N)^T$ is the vector of unknowns, $\mathbf{f}=(f(x_1)+\alpha/h^2,f(x_2),\dots,f(x_{N-1}),f(x_N)+\beta/h^2)^T$, and A is the tridiagonal matrix

$$\mathrm{A}=\mathrm{tridiag}(-1,2,-1)=\begin{bmatrix} 2 & -1 & 0 & \dots & 0\\ -1 & 2 & \ddots & & \vdots\\ 0 & \ddots & \ddots & -1 & 0\\ \vdots & & -1 & 2 & -1\\ 0 & \dots & 0 & -1 & 2\end{bmatrix}. \tag{9.15}$$

This system admits a unique solution since A is symmetric and positive definite (see Exercise 9.1). Moreover, it can be solved by the Thomas algorithm introduced in Section 5.6. We note however that, for small values of h (and thus for large values of N), A is ill-conditioned. Indeed, $K(\mathrm{A})=\lambda_{max}(\mathrm{A})/\lambda_{min}(\mathrm{A})=Ch^{-2}$, for a suitable constant C independent of h (see Exercise 9.2). Consequently, the numerical solution of system (9.14), by either direct or iterative methods, requires special care. In particular, when using iterative methods a suitable preconditioner ought to be employed.

It is possible to prove (see, e.g., [QSS07, Chapter 12]) that if $f\in C^2([a,b])$ then

$$\boxed{\max_{j=0,\dots,N+1}|u(x_j)-u_j|\le\frac{(b-a)^2h^2}{96}\max_{x\in[a,b]}|f''(x)|} \tag{9.16}$$

that is, the finite difference method (9.13) converges with order two with respect to h.

In Program 9.1 we solve the following boundary-value problem (the so-called *diffusion-convection-reaction problem*)

$$\begin{cases}-\mu u''(x)+\eta u'(x)+\sigma u(x)=f(x) & \text{for } x\in(a,b),\\ u(a)=\alpha & u(b)=\beta,\end{cases} \tag{9.17}$$

$\mu>0$, η and $\sigma>0$ constants, which is a generalization of problem (9.10).

For this problem the finite difference method, which generalizes (9.13), reads:

$$\begin{cases} -\mu\dfrac{u_{j+1}-2u_j+u_{j-1}}{h^2}+\eta\dfrac{u_{j+1}-u_{j-1}}{2h}+\sigma u_j=f(x_j), & j=1,\ldots,N,\\ u_0=\alpha, & u_{N+1}=\beta. \end{cases}$$

The input parameters of Program 9.1 are the end-points `a` and `b` of the interval, the number `N` of internal nodes, the constant coefficients μ, η and σ and the function handle `bvpfun` associated with the function $f(x)$. Finally, `ua` and `ub` represent the values that the solution should attain at `x=a` and `x=b`, respectively. Output parameters are the vector of nodes `xh` and the computed solution `uh`. Notice that the solutions can be affected by spurious oscillations if $h \geq 2\mu/\eta$ (see next Section).

Program 9.1. bvp: approximation of a two-point diffusion-convection-reaction problem by the finite difference method

```
function [xh,uh]=bvp(a,b,N,mu,eta,sigma,bvpfun,...
                    ua,ub,varargin)
%BVP  Solves two-point boundary value problems.
%  [XH,UH]=BVP(A,B,N,MU,ETA,SIGMA,BVPFUN,UA,UB)
%  solves the boundary-value problem
%     -MU*D(DU/DX)/DX+ETA*DU/DX+SIGMA*U=BVPFUN
%  on the interval (A,B) with boundary conditions
%  U(A)=UA and U(B)=UB, by the centered finite
%  difference method at N equispaced nodes
%  internal to (A,B). BVPFUN is a function handle.
%  [XH,UH]=BVP(A,B,N,MU,ETA,SIGMA,BVPFUN,UA,UB,...
%  P1,P2,...) passes the additional parameters
%  P1, P2, ... to the function BVPFUN.
%  XH contains the nodes of the discretization,
%  including the boundary nodes.
%  UH contains the numerical solutions.
h = (b-a)/(N+1);
xh = (linspace(a,b,N+2))';
hm = mu/h^2;
hd = eta/(2*h);
e =ones(N,1);
A = spdiags([-hm*e-hd (2*hm+sigma)*e -hm*e+hd],...
    -1:1, N, N);
xi = xh(2:end-1);
f =bvpfun(xi,varargin{:});
f(1) =  f(1)+ua*(hm+hd);
f(end) = f(end)+ub*(hm-hd);
uh = A\f;
uh=[ua; uh; ub];
return
```

9.2.2 Finite difference approximation of a convection-dominated problem

We consider now the following generalization of the boundary-value problem (9.10)

$$\boxed{\begin{array}{ll} -\mu u''(x) + \eta u'(x) = f(x) & \text{for } x \in (a,b), \\ u(a) = \alpha, & u(b) = \beta, \end{array}} \tag{9.18}$$

μ and η being positive constants. This is the so-called *convection-diffusion problem* since the terms $-\mu u''(x)$ and $\eta u'(x)$ are responsible of diffusion and convection of the unknown function $u(x)$, respectively. The *global Péclet number*, associated to equation (9.18), is defined as

$$\mathbb{P}\mathrm{e}_{gl} = \frac{\eta(b-a)}{2\mu}, \tag{9.19}$$

and it provides a measure of how much the convective term prevails over the diffusive one. A problem featuring $\mathbb{P}\mathrm{e}_{gl} \gg 1$ will be named *convection-dominated problem.*

A possible discretization of (9.18) reads

$$\begin{cases} -\mu \dfrac{u_{j+1} - 2u_j + u_{j-1}}{h^2} + \eta \dfrac{u_{j+1} - u_{j-1}}{2h} = f(x_j),\ j = 1, \ldots, N, \\ u_0 = \alpha, \quad u_{N+1} = \beta, \end{cases} \tag{9.20}$$

in which the centered finite difference scheme (4.9) has been used to approximate the convection term. As for the Poisson equation, one can prove that the error between the solution of the discrete problem (9.20) and that of the continuous problem (9.18) satisfies the following estimate

$$\max_{j=0,\ldots,N+1} |u(x_j) - u_j| \le Ch^2 \max_{x\in[a,b]} |f''(x)|. \tag{9.21}$$

The constant C is proportional to $\mathbb{P}\mathrm{e}_{gl}$, therefore it is very large when the convection dominates the diffusion. Thus, if the discretization step h is not small enough, the numerical solution computed by the scheme (9.20) may be highly inaccurate and exhibit strong oscillations which are far from satisfying the continuous problem. For a more detailed analysis of this phenomenon we introduce the so-called *local Péclet number* (also named "grid" Péclet number)

$$\mathbb{P}\mathrm{e} = \frac{\eta h}{2\mu}. \tag{9.22}$$

One can prove that the solution of the discrete problem (9.20) does not exhibit oscillations if $\mathbb{P}\mathrm{e} < 1$ (see [Qua13, Chap. 5]). Thus, in order to

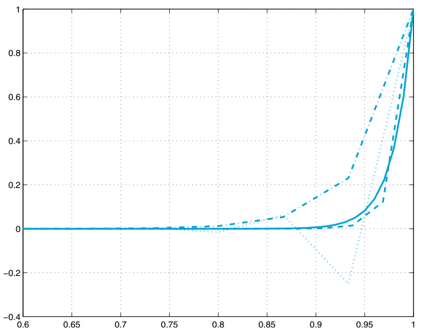


Figure 9.3. Exact solution (*solid line*), centered finite difference approximation with $h = 1/15$ ($\mathbb{P}\mathrm{e} > 1$) (*dotted line*), centered finite difference approximation with $h = 1/32$ ($\mathbb{P}\mathrm{e} < 1$) (*dashed line*), upwind finite difference approximation with $h = 1/15$ (*dashed-dotted line*) of the solution of problem (9.18) with $a = 0$, $b = 1$, $\alpha = 0$, $\beta = 1$, $f(x) = 0$, $\mu = 1/50$ and $\eta = 1$. For clearness, numerical solutions have been plotted on the interval $[0.6, 1]$ instead of $[0, 1]$

ensure a good numerical solution, we have to choose a discretization step $h < 2\mu/\eta$. Unfortunately, such a choice is not convenient when the ratio $2\mu/\eta$ is very small.

A possible alternative consists in choosing a different approximation of the convective term u'; precisely, instead to use the centered finite difference (4.9), we can employ the backward finite difference (4.8), so that the system (9.20) is replaced by

$$\begin{cases} -\mu \dfrac{u_{j+1} - 2u_j + u_{j-1}}{h^2} + \eta \dfrac{u_j - u_{j-1}}{h} = f(x_j), \quad j = 1, \ldots, N, \\ u_0 = \alpha, \qquad\qquad\qquad\qquad\qquad\qquad u_{N+1} = \beta, \end{cases} \tag{9.23}$$

which is known as *upwind* scheme. It is possible to prove that if (9.18) is approximated by (9.23), then the yielded numerical solution will not exhibit any oscillation, as the graphs reported in Figure 9.3 confirm.

9.2.3 Finite element approximation of the one-dimensional Poisson problem

The *finite element method* represents an alternative to the finite difference method for the approximation of boundary-value problems and is derived from a suitable reformulation of the differential problem (9.10).

Let us consider again (9.10) and multiply both sides of the differential equation by a generic function $v \in C^1([a, b])$. Integrating the corresponding equality on the interval (a, b) and using integration by parts we obtain

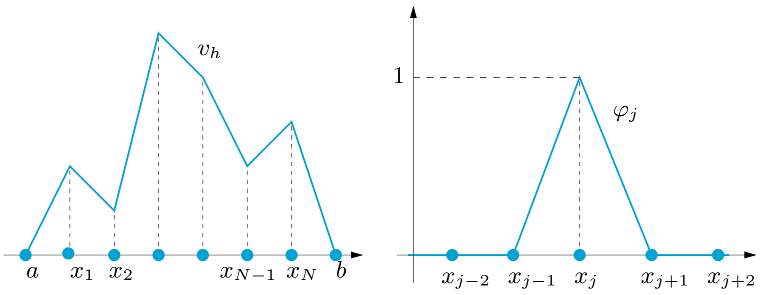


Figure 9.4. At left, a generic function $v_h \in V_h^0$. At right, the basis function of V_h^0 associated with the jth node

$$\int_a^b u'(x)v'(x)\ dx - [u'(x)v(x)]_a^b = \int_a^b f(x)v(x)\ dx.$$

By making the further assumption that v vanishes at the end-points $x = a$ and $x = b$, problem (9.10) becomes: find $u \in C^1([a,b])$ such that $u(a) = \alpha$, $u(b) = \beta$ and

$$\int_a^b u'(x)v'(x)\ dx = \int_a^b f(x)v(x)\ dx \tag{9.24}$$

for each $v \in C^1([a,b])$ such that $v(a) = v(b) = 0$. This is called *weak formulation* of problem (9.10). (Indeed, both u and the test function v can be less regular than $C^1([a,b])$, see, e.g. [Qua13], [QSS07], [QV94].)

Its finite element approximation is defined as follows:

find $u_h \in V_h$ such that $u_h(a) = \alpha, u_h(b) = \beta$ and

$$\sum_{j=0}^{N} \int_{x_j}^{x_{j+1}} u_h'(x)v_h'(x)\ dx = \int_a^b f(x)v_h(x)\ dx, \qquad \forall v_h \in V_h^0 \tag{9.25}$$

where

$$V_h = \left\{v_h \in C^0([a,b]) :\ v_{h|I_j} \in \mathbb{P}_1, j = 0,\dots,N\right\}, \tag{9.26}$$

i.e. V_h is the space of continuous functions on $[a,b]$ whose restrictions on every sub-interval I_j are linear polynomials. Moreover, V_h^0 is the subspace of V_h of those functions vanishing at the end-points a and b. V_h is called space of finite-elements of degree 1.

The functions in V_h^0 are piecewise linear polynomials (see Figure 9.4, left). In particular, every function v_h of V_h^0 admits the representation

$$v_h(x) = \sum_{j=1}^{N} v_h(x_j)\varphi_j(x),$$

where for $j = 1, \ldots, N$,

$$\varphi_j(x) = \begin{cases} \dfrac{x - x_{j-1}}{x_j - x_{j-1}} & \text{if } x \in I_{j-1}, \\ \dfrac{x - x_{j+1}}{x_j - x_{j+1}} & \text{if } x \in I_j, \\ 0 & \text{otherwise.} \end{cases}$$

Thus, φ_j is null at every node x_i except at x_j where $\varphi_j(x_j) = 1$ (see Figure 9.4, right). The functions φ_j, $j = 1, \ldots, N$ are called *shape functions* and provide a basis for the vector space V_h^0.

Consequently, to fulfill (9.25) for any function in V_h is equivalent to fulfill it only for the shape functions φ_j, $j = 1, \ldots, N$. By exploiting the fact that φ_j vanishes outside the intervals I_{j-1} and I_j, from (9.25) we obtain

$$\int\limits_{I_{j-1}\cup I_j} u_h'(x)\varphi_j'(x)\ dx = \int\limits_{I_{j-1}\cup I_j} f(x)\varphi_j(x)\ dx, \quad j = 1, \ldots, N. \quad (9.27)$$

On the other hand, we can write $u_h(x) = \sum_{j=1}^{N} u_j\varphi_j(x) + \alpha\varphi_0(x) + \beta\varphi_{N+1}(x)$, where $u_j = u_h(x_j)$, $\varphi_0(x) = (x_1 - x)/(x_1 - a)$ for $a \le x \le x_1$, and $\varphi_{N+1}(x) = (x - x_N)/(b - x_N)$ for $x_N \le x \le b$, while both $\varphi_0(x)$ and $\varphi_{N+1}(x)$ are zero otherwise. By substituting this expression in (9.27), we find:

$$u_1 \int\limits_{I_0\cup I_1} \varphi_1'(x)\varphi_1'(x)\ dx + u_2 \int\limits_{I_1} \varphi_2'(x)\varphi_1'(x)\ dx = \int\limits_{I_0\cup I_1} f(x)\varphi_1(x)\ dx + \frac{\alpha}{x_1 - a},$$

$$u_{j-1} \int\limits_{I_{j-1}} \varphi_{j-1}'(x)\varphi_j'(x)\ dx + u_j \int\limits_{I_{j-1}\cup I_j} \varphi_j'(x)\varphi_j'(x)\ dx + u_{j+1} \int\limits_{I_j} \varphi_{j+1}'(x)\varphi_j'(x)\ dx = \int\limits_{I_{j-1}\cup I_j} f(x)\varphi_j(x)\ dx, \quad j = 2, \ldots, N-1,$$

$$u_{N-1} \int\limits_{I_{N-1}} \varphi_{N-1}'(x)\varphi_N'(x)\ dx + u_N \int\limits_{I_{N-1}\cup I_N} \varphi_N'(x)\varphi_N'(x)\ dx = \int\limits_{I_{N-1}\cup I_N} f(x)\varphi_j(x)\ dx + \frac{\beta}{b - x_N}.$$

In the special case where all intervals have the same length h, then $\varphi'_{j-1} = -1/h$ in I_{j-1}, $\varphi'_j = 1/h$ in I_{j-1} and $\varphi'_j = -1/h$ in I_j, $\varphi'_{j+1} = 1/h$ in I_j. Consequently, we obtain

$$\begin{aligned}
2u_1 - u_2 &= h \int\limits_{I_0 \cup I_1} f(x)\varphi_1(x)\ dx + \alpha,\\
-u_{j-1} + 2u_j - u_{j+1} &= h \int\limits_{I_{j-1} \cup I_j} f(x)\varphi_j(x)\ dx, \qquad j = 2, \ldots, N-1,\\
-u_{N-1} + 2u_N &= h \int\limits_{I_{N-1} \cup I_N} f(x)\varphi_N(x)\ dx + \beta.
\end{aligned}$$

The yielded linear system has unknowns $\{u_1, \ldots, u_N\}$ and shares the same matrix (9.15) as the finite difference system, however it has a different right-hand side (and a different solution too, in spite of coincidence of notation). Finite difference and finite element solutions share however the same accuracy with respect to h when the nodal maximum error is computed.

We notice that 2nd-order convergence with respect to h is guaranteed for finite difference approximation if $f \in C^2([a, b])$ (see (9.21)), while for finite elements it is sufficient that f be a square-integrable function in (a, b), i.e.,

$$\int_a^b f^2(x)dx < +\infty.$$

Obviously the finite element approach can be generalized to problems like (9.17) (also in the case when μ, η and σ depend on x) and (9.18).

To approximate the convection-dominated problem (9.18), the upwind scheme used for finite differences can be reproduced also for finite-elements. More precisely, by noting that

$$\frac{u_i - u_{i-1}}{h} = \frac{u_{i+1} - u_{i-1}}{2h} - \frac{h}{2}\frac{u_{i+1} - 2u_i + u_{i-1}}{h^2},$$

we can conclude that decentralizing finite differences is equivalent to perturb the centered incremental ratio by a term corresponding to a second-order derivative. This additional term can be interpreted as an *artificial viscosity*. In other words, using upwind with finite-elements is equivalent to solve, by the (centered) Galerkin method, the following perturbed problem

$$-\mu_h u''(x) + \eta u'(x) = f(x), \tag{9.28}$$

where $\mu_h = (1 + \mathbb{P}\mathrm{e})\mu$ is the augmented viscosity.

A further generalization of linear finite element methods consists of using piecewise polynomials of degree greater than 1, allowing the achievement of higher convergence orders. In these cases, the finite element matrix does not coincide anymore with that of finite differences.

See Exercises 9.1-9.7.

9.2.4 Finite difference approximation of the two-dimensional Poisson problem

Let us consider the Poisson problem (9.2), in a two-dimensional region Ω.

The idea behind finite differences relies on approximating the partial derivatives that are present in the PDE again by incremental ratios computed on a suitable grid (called the computational grid) made of a finite number of nodes. Then the solution u of the PDE will be approximated only at these nodes.

The first step therefore consists of introducing a computational grid. Assume for simplicity that Ω is the rectangle $(a,b)\times(c,d)$. Let us introduce a partition of $[a,b]$ in subintervals (x_i,x_{i+1}) for $i=0,\ldots,N_x$, with $x_0=a$ and $x_{N_x+1}=b$. Let us denote by $\Delta_x=\{x_0,\ldots,x_{N_x+1}\}$ the set of end-points of such intervals and by $h_x=\max\limits_{i=0,\ldots,N_x}(x_{i+1}-x_i)$ their maximum length.

In a similar manner we introduce a discretization of the y-axis $\Delta_y=\{y_0,\ldots,y_{N_y+1}\}$ with $y_0=c$, $y_{N_y+1}=d$ and $h_y=\max\limits_{j=0,\ldots,N_y}(y_{j+1}-y_j)$. The cartesian product $\Delta_h=\Delta_x\times\Delta_y$ provides the computational grid on Ω (see Figure 9.5), and $h=\max\{h_x,h_y\}$ is a characteristic measure of the grid-size. We are looking for values $u_{i,j}$ which approximate $u(x_i,y_j)$. We will assume for the sake of simplicity that the nodes be uniformly spaced, that is, $x_i=x_0+ih_x$ for $i=0,\ldots,N_x+1$ and $y_j=y_0+jh_y$ for $j=0,\ldots,N_y+1$.

The second order partial derivatives of a function can be approximated by a suitable incremental ratio, as we did for ordinary derivatives. In the case of a function of two variables, we define the following incremental ratios:

$$\begin{aligned}\delta_x^2 u_{i,j}&=\frac{u_{i-1,j}-2u_{i,j}+u_{i+1,j}}{h_x^2},\\ \delta_y^2 u_{i,j}&=\frac{u_{i,j-1}-2u_{i,j}+u_{i,j+1}}{h_y^2}.\end{aligned}\tag{9.29}$$

They are second order accurate with respect to h_x and h_y, respectively, for the approximation of $\partial^2u/\partial x^2$ and $\partial^2u/\partial y^2$ at the node (x_i,y_j). If we replace the second order partial derivatives of u with the formula

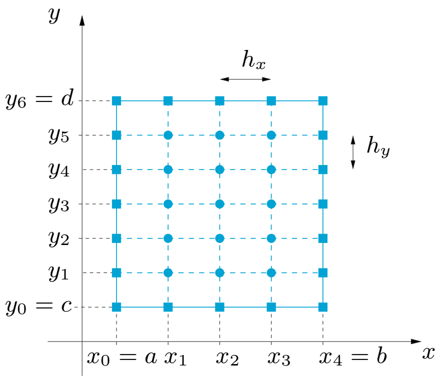


Figure 9.5. The computational grid $\varDelta_h$ with only 15 internal nodes on a rectangular domain

(9.29), by requiring that the PDE is satisfied at all internal nodes of $\varDelta_h$, we obtain the following set of equations:

$$-(\delta_x^2 u_{i,j} + \delta_y^2 u_{i,j}) = f_{i,j}, \quad i = 1, \ldots, N_x,\ j = 1, \ldots, N_y. \tag{9.30}$$

We have set $f_{i,j} = f(x_i, y_j)$. We must add the equations that enforce the Dirichlet data at the boundary, which are

$$u_{i,j} = g_{i,j} \quad \forall i, j \text{ such that } (x_i, y_j) \in \partial\varDelta_h, \tag{9.31}$$

where $\partial\varDelta_h$ indicates the set of nodes belonging to the boundary $\partial\varOmega$ of $\varOmega$. These nodes are indicated by small squares in Figure 9.5. If we make the further assumption that the computational grid is uniform in both cartesian directions, that is, $h_x = h_y = h$, instead of (9.30) we obtain

$$\boxed{\begin{aligned} -\frac{1}{h^2}(u_{i-1,j} + u_{i,j-1} - 4u_{i,j} + u_{i,j+1} + u_{i+1,j}) = f_{i,j}, \\ i = 1, \ldots, N_x,\ j = 1, \ldots, N_y \end{aligned}} \tag{9.32}$$

The system given by equations (9.32) (or (9.30)) and (9.31) allows the computation of the nodal values $u_{i,j}$ at all nodes of $\varDelta_h$. For every fixed pair of indices i and j, equation (9.32) involves five unknown nodal values as we can see in Figure 9.6. For that reason this finite difference scheme is called *the five-point scheme* for the Laplace operator. We note that the unknowns associated with the boundary nodes can be eliminated using (9.31) and therefore (9.30) (or (9.32)) involves only $N = N_x N_y$ unknowns.

The resulting system can be written in a more interesting form if we adopt the *lexicographic* order according to which the nodes (and,

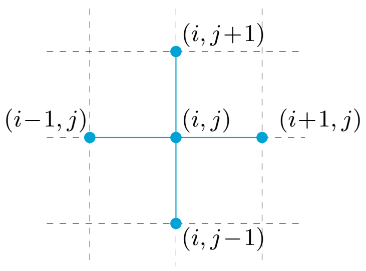


Figure 9.6. The stencil of the five point scheme for the Laplace operator

correspondingly, the unknown components) are numbered by proceeding from left to right and from the bottom to the top. By so doing, we obtain a system like (9.14), with a matrix $\mathrm{A} \in \mathbb{R}^{N \times N}$ which takes the following block tridiagonal form:

$$\mathrm{A} = \mathrm{tridiag}(\mathrm{D}, \mathrm{T}, \mathrm{D}). \tag{9.33}$$

There are N_y rows and N_y columns, and every entry (denoted by a capital letter) consists of a $N_x \times N_x$ matrix. In particular, $\mathrm{D} \in \mathbb{R}^{N_x \times N_x}$ is a diagonal matrix whose diagonal entries are $-1/h_y^2$, while $\mathrm{T} \in \mathbb{R}^{N_x \times N_x}$ is a symmetric tridiagonal matrix

$$\mathrm{T} = \mathrm{tridiag}(-\frac{1}{h_x^2}, \frac{2}{h_x^2} + \frac{2}{h_y^2}, -\frac{1}{h_x^2}).$$

A is symmetric since all diagonal blocks are symmetric. It is also positive definite, that is $\mathbf{v}^T \mathrm{A} \mathbf{v} > 0 \ \forall \mathbf{v} \in \mathbb{R}^N$, $\mathbf{v} \neq \mathbf{0}$. Actually, by partitioning $\mathbf{v}$ in N_y vectors $\mathbf{v}_k$ of length N_x we obtain

$$\mathbf{v}^T \mathrm{A} \mathbf{v} = \sum_{k=1}^{N_y} \mathbf{v}_k^T \mathrm{T} \mathbf{v}_k - \frac{2}{h_y^2} \sum_{k=1}^{N_y - 1} \mathbf{v}_k^T \mathbf{v}_{k+1}. \tag{9.34}$$

We can write $\mathrm{T} = 2/h_y^2 \mathrm{I} + 1/h_x^2 \mathrm{K}$ where K is the (symmetric and positive definite) matrix given in (9.15) and I is the identity matrix. Consequently, using the identity $2a(a-b) = a^2 - b^2 + (a-b)^2$ and some algebraic manipulation, (9.34) reads

$$\begin{aligned}\mathbf{v}^T \mathrm{A} \mathbf{v} = &\frac{1}{h_x^2} \sum_{k=1}^{N_y - 1} \mathbf{v}_k^T \mathrm{K} \mathbf{v}_k \\ &+ \frac{1}{h_y^2} \left(\mathbf{v}_1^T \mathbf{v}_1 + \mathbf{v}_{N_y}^T \mathbf{v}_{N_y} + \sum_{k=1}^{N_y - 1} (\mathbf{v}_k - \mathbf{v}_{k+1})^T (\mathbf{v}_k - \mathbf{v}_{k+1}) \right),\end{aligned}$$

which is a strictly positive real number since K is positive definite and at least one vector $\mathbf{v}_k$ is non-null.

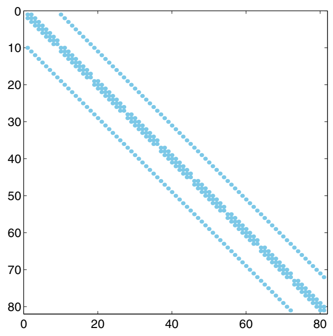


Figure 9.7. Pattern of the matrix associated with the five-point scheme using the lexicographic ordering of the unknowns

Having proven that A is non-singular we can conclude that the finite difference system admits a unique solution $\mathbf{u}_h$.

The matrix A is *sparse*; as such, it will be stored in the format `sparse` of MATLAB (see Section 5.3). In Figure 9.7 (obtained by using the command `spy(A)`) we report the structure of the matrix corresponding to a uniform grid of 11×11 nodes, after having eliminated the rows and columns associated to the nodes of $\partial\Delta_h$. It can be noted that the only nonzero elements lie on five diagonals.

Since A is symmetric and positive definite, the associated system can be solved efficiently by either direct or iterative methods, as illustrated in Chapter 5. Finally, it is worth pointing out that A shares with its one-dimensional analog the property of being ill-conditioned: indeed, its condition number grows like h^{-2} as h tends to zero.

In the Program 9.2 we construct and solve the system (9.30)-(9.31) (using the command \, see Section 5.8). The input parameters `a`, `b`, `c` and `d` denote the endpoints of the intervals generating the domain $\Omega = (a, b) \times (c, d)$, while `nx` and `ny` denote the values of N_x and N_y (the case $N_x \neq N_y$ is admitted). Finally, the two function handles `fun` and `bound` are associated with the right-hand side $f = f(x, y)$ (otherwise called the source term) and the Dirichlet boundary data $g = g(x, y)$, respectively. The output variable `uh` is a matrix whose j, ith entry is $u_{i,j}$, while `xh` and `yh` are vectors whose components are the nodes x_i and y_j, respectively, all including the nodes of the boundary. The numerical

`mesh` solution can be visualized by the command `mesh(x,y,u)`. The (optional) input function `uex` stands for the exact solution of the original problem for those cases (of theoretical interest) where this solution is known. In such cases the output parameter `error` contains the nodal relative error between the exact and numerical solution, which is computed as follows:

$$\texttt{error} = \max_{i,j} |u(x_i, y_j) - u_{i,j}| / \max_{i,j} |u(x_i, y_j)|.$$

Program 9.2. poissonfd: approximation of the Poisson problem with Dirichlet boundary data by the five-point finite difference method

```
function [xh,yh,uh,error]=poissonfd(a,b,c,d,nx,ny,...
                                   fun,bound,uex,varargin)
%POISSONFD two-dimensional Poisson solver
%   [XH,YH,UH]=POISSONFD(A,B,C,D,NX,NY,FUN,BOUND) solves
%   by the five-point finite difference scheme the
%   problem -LAPL(U) = FUN in the rectangle (A,B)X(C,D)
%   with Dirichlet boundary conditions U(X,Y)=BOUND(X,Y)
%   at any (X,Y) on the boundary of the rectangle.
%   [XH,YH,UH,ERROR]=POISSONFD(A,B,C,D,NX,NY,FUN,...
%   BOUND,UEX) computes also the maximum nodal error
%   ERROR with respect to the exact solution UEX.
%   FUN,BOUND and UEX are function handles.
%   [XH,YH,UH,ERROR]=POISSONFD(A,B,C,D,NX,NY,FUN,...
%   BOUND,UEX,P1,P2, ...) passes the optional arguments
%   P1,P2,... to the functions FUN,BOUND,UEX.
if nargin == 8
    uex = @(x,y)0+0*x+0*y;
end
nx1 = nx+2; ny1=ny+2; dim = nx1*ny1;
hx = (b-a)/(nx+1); hy = (d-c)/(ny+1);
    hx2 = hx^2;      hy2 = hy^2;
kii = 2/hx2+2/hy2; kix = -1/hx2;  kiy = -1/hy2;
K = speye(dim,dim);  rhs = zeros(dim,1);
y = c;
for m = 2:ny+1
 x = a; y = y + hy;
 for n = 2:nx+1
   i = n+(m-1)*nx1; x = x + hx;
   rhs(i) = fun(x,y,varargin{:});
   K(i,i) = kii; K(i,i-1) = kix; K(i,i+1) = kix;
   K(i,i+nx1) = kiy;   K(i,i-nx1) = kiy;
 end
end
rhs1 = zeros(dim,1); xh = [a:hx:b]'; yh = [c:hy:d];
rhs1(1:nx1) = bound(xh,c,varargin{:});
rhs1(dim-nx-1:dim) = bound(xh,d,varargin{:});
rhs1(1:nx1:dim-nx-1) = bound(a,yh,varargin{:});
rhs1(nx1:nx1:dim) = bound(b,yh,varargin{:});
rhs = rhs - K*rhs1;
nbound = [[1:nx1],[dim-nx-1:dim],[1:nx1:dim-nx-1],...
    [nx1:nx1:dim]];
ninternal = setdiff([1:dim],nbound);
K = K(ninternal,ninternal);
rhs = rhs(ninternal);
utemp = K\ rhs;
u = rhs1; u (ninternal) = utemp;
k = 1; y = c;
for j = 1:ny1
    x = a;
    for i = 1:nx1
        uh(j,i) = u(k);          k = k + 1;
        ue(j,i) = uex(x,y,varargin{:});
        x = x + hx;
    end
    y = y + hy;
```

```
end
if nargout == 4 & nargin >= 9
    error = max(max(abs(uh-ue)))/max(max(abs(ue)));
elseif nargout == 4 & nargin ==8
    warning('Exact solution not available');
    error = [ ];
end
end
```

Example 9.1 The transverse displacement u of an elastic membrane from the reference plane $z = 0$, under a load whose intensity is $f(x, y) = 8\pi^2 \sin(2\pi x) \cos(2\pi y)$, satisfies a Poisson problem like (9.2) in the domain $\Omega = (0, 1)^2$. The Dirichlet value of the displacement is prescribed on $\partial\Omega$ as follows: $g = 0$ on the sides $x = 0$ and $x = 1$, and $g(x, 0) = g(x, 1) = \sin(2\pi x)$, $0 < x < 1$. This problem admits the exact solution $u(x, y) = \sin(2\pi x) \cos(2\pi y)$. In Figure 9.8 we show the numerical solution obtained by the five-point finite difference scheme on a uniform grid. Two different values of h have been used: $h = 1/10$ (*left*) and $h = 1/20$ (*right*). When h decreases the numerical solution improves, and actually the nodal relative error is 0.0292 for $h = 1/10$ and 0.0081 for $h = 1/20$. ■

Also the finite element method can be easily extended to the two-dimensional case. To this end the problem (9.2) must be reformulated in an integral form and the partition of the interval (a, b) in one dimension must be replaced by a decomposition of Ω by polygons (typically, triangles) called *elements*. The generic shape function φ_k will still be a continuous function, whose restriction on each element is a polynomial of degree 1 on each element, which is equal to 1 at the kth vertex (or node) of the triangulation and 0 at all other vertices. For its implementation one can use the MATLAB toolbox pde.

pde

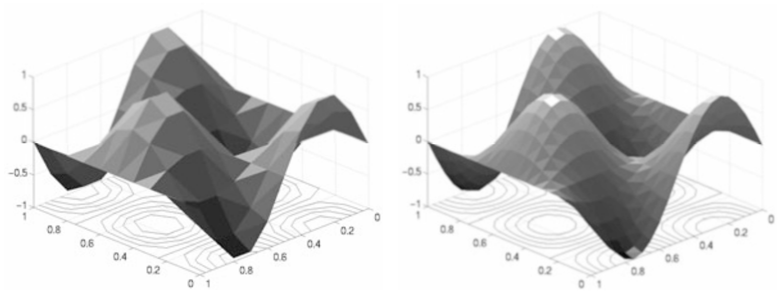


Figure 9.8. Transverse displacement of an elastic membrane computed on two uniform grids, coarser at left and finer at right. On the horizontal plane we report the isolines of the numerical solution. The triangular partition of Ω only serves the purpose of the visualization of the results

9.2.5 Consistency and convergence of finite difference discretization of the Poisson problem

In the previous section we have shown that the solution of the finite difference problem exists and is unique. Now we investigate the approximation error. We will assume for simplicity that $h_x = h_y = h$. If

$$\boxed{\max_{i,j} |u(x_i, y_j) - u_{i,j}| \to 0 \ \text{ as } h \to 0} \tag{9.35}$$

the method used to compute $u_{i,j}$ is called convergent.

As we have already pointed out (see Remark 8.1), consistency is a necessary condition for convergence. A method is *consistent* if the residual, that is the error obtained when the exact solution is plugged into the numerical scheme, tends to zero when h tends to zero. If we consider the five point finite difference scheme, at every internal node (x_i, y_j) of $\varDelta_h$ we define

$$\tau_h(x_i, y_j) = -f(x_i, y_j)$$
$$-\frac{1}{h^2}\left[u(x_{i-1}, y_j) + u(x_i, y_{j-1}) - 4u(x_i, y_j) + u(x_i, y_{j+1}) + u(x_{i+1}, y_j)\right].$$

This is the *local truncation error* at the node (x_i, y_j). By (9.2) we obtain

$$\begin{aligned}\tau_h(x_i, y_j) = &\left\{\frac{\partial^2 u}{\partial x^2}(x_i, y_j) - \frac{u(x_{i-1}, y_j) - 2u(x_i, y_j) + u(x_{i+1}, y_j)}{h^2}\right\}\\ &+\left\{\frac{\partial^2 u}{\partial y^2}(x_i, y_j) - \frac{u(x_i, y_{j-1}) - 2u(x_i, y_j) + u(x_i, y_{j+1})}{h^2}\right\}.\end{aligned}$$

Thanks to the analysis that was carried out in Section 9.2.4 we can conclude that both terms vanish as h tends to 0. Thus

$$\lim_{h\to 0} \tau_h(x_i, y_j) = 0, \qquad (x_i, y_j) \in \varDelta_h \setminus \partial\varDelta_h,$$

that is, the five-point method is consistent.

It is also convergent, as stated in the following Proposition (for its proof, see, e.g., [IK66]):

Proposition 9.1 *Assume that the exact solution $u \in C^4(\bar{\Omega})$, i.e. all its partial derivatives up to the fourth order are continuous in the closed domain $\bar{\Omega}$. Then there exists a constant $C > 0$ such that*

$$\boxed{\max_{i,j} |u(x_i, y_j) - u_{i,j}| \le CMh^2} \tag{9.36}$$

where M is the maximum absolute value attained by the fourth order derivatives of u in $\bar{\Omega}$.

Example 9.2 Let us experimentally verify that the five-point scheme applied to solve the Poisson problem of Example 9.1 converges with order two with respect to h. We start from $h = 1/4$ and, then we halve subsequently the value of h, until $h = 1/64$, through the following instructions:

```
a=0;b=1;c=0;d=1;
f=@(x,y) 8*pi^2*sin(2*pi*x).*cos(2*pi*y);

g=@(x,y) sin(2*pi*x).*cos(2*pi*y);
uex=g; nx=4; ny=4;
for n=1:5
   [xh,yh,uh,error(n)]=poissonfd(a,b,c,d,nx,ny,f,g,uex);
     nx = 2*nx; ny = 2*ny;
```

The vector containing the error is

```
format short e; error
   1.3565e-01   4.3393e-02   1.2308e-02   3.2775e-03   8.4557e-04
```

As we can verify using the following commands (see formula (1.12))

```
log(abs(error(1:end-1)./error(2:end)))/log(2)
  1.6443e+00   1.8179e+00   1.9089e+00   1.9546e+00
```

this error decreases as h^2 when $h \to 0$. ■

9.2.6 Finite difference approximation of the one-dimensional heat equation

We consider the one-dimensional heat equation (9.4) with homogeneous Dirichlet boundary conditions $u(a,t) = u(b,t) = 0$ for any $t > 0$ and initial condition $u(x,0) = u^0(x)$ for $x \in [a,b]$.

To solve this equation numerically we have to discretize both the x and t variables. We can start by dealing with the x-variable, following the same approach as in Section 9.2.1. We denote by $u_j(t)$ an approximation of $u(x_j,t)$, $j = 0,\dots,N+1$, and approximate the Dirichlet problem (9.4) by the scheme: for all $t > 0$

$$\begin{cases} \dfrac{du_j}{dt}(t) - \dfrac{\mu}{h^2}(u_{j-1}(t) - 2u_j(t) + u_{j+1}(t)) = f_j(t), \quad j = 1,\dots,N, \\ u_0(t) = u_{N+1}(t) = 0, \end{cases}$$

where $f_j(t) = f(x_j,t)$ and, for $t = 0$,

$$u_j(0) = u^0(x_j), \qquad j = 0,\dots,N+1.$$

This is actually a *semi-discretization* of the heat equation, yielding a system of ordinary differential equations of the following form

$$\begin{cases} \dfrac{d\mathbf{u}}{dt}(t) = -\dfrac{\mu}{h^2}\mathrm{A}\mathbf{u}(t) + \mathbf{f}(t) \qquad \forall t > 0, \\ \mathbf{u}(0) = \mathbf{u}^0, \end{cases} \tag{9.37}$$

where $\mathbf{u}(t) = (u_1(t), \ldots, u_N(t))^T$ is the vector of unknowns, $\mathbf{f}(t) = (f_1(t), \ldots, f_N(t))^T$, $\mathbf{u}^0 = (u^0(x_1), \ldots, u^0(x_N))^T$, and A is the tridiagonal matrix introduced in (9.15). Note that for the derivation of (9.37) we have assumed that $u^0(x_0) = u^0(x_{N+1}) = 0$, which is coherent with the homogeneous Dirichlet boundary conditions.

A popular scheme for the integration in time of (9.37) is the so-called *$\theta-$method*. Let $\Delta t > 0$ be a constant time-step, and denote by v^k the value of a variable v referred at the time level $t^k = k\Delta t$. Then the θ-method reads

$$\boxed{\begin{aligned} &\frac{\mathbf{u}^{k+1}-\mathbf{u}^k}{\Delta t} = -\frac{\mu}{h^2}\mathrm{A}(\theta\mathbf{u}^{k+1} + (1-\theta)\mathbf{u}^k) + \theta\mathbf{f}^{k+1} + (1-\theta)\mathbf{f}^k, \\ &\hspace{20em} k = 0, 1, \ldots \\ &\mathbf{u}^0 \text{ given} \end{aligned}} \tag{9.38}$$

or, equivalently,

$$\left(\mathrm{I} + \frac{\mu}{h^2}\theta\Delta t\mathrm{A}\right)\mathbf{u}^{k+1} = \left(\mathrm{I} - \frac{\mu}{h^2}\Delta t(1-\theta)\mathrm{A}\right)\mathbf{u}^k + \mathbf{g}^{k+1}, \tag{9.39}$$

where $\mathbf{g}^{k+1} = \Delta t(\theta\mathbf{f}^{k+1} + (1-\theta)\mathbf{f}^k)$ and I is the identity matrix of order N.

For suitable values of the parameter θ, from (9.39) we can recover some familiar methods that have been introduced in Chapter 8. For example, if $\theta = 0$ the method (9.39) coincides with the forward Euler scheme and we can obtain $\mathbf{u}^{k+1}$ explicitly; otherwise, a linear system (with constant matrix $\mathrm{I} + \mu\theta\Delta t\mathrm{A}/h^2$) needs to be solved at each time level.

Regarding stability, when $f = 0$ the exact solution $u(x,t)$ tends to zero for every x as $t \to \infty$. Then we would expect the discrete solution to have the same behavior, in which case we would call our scheme (9.39) *asymptotically stable*, this being coherent with the absolute stability concept defined in Section 8.6 for ordinary differential equations.

In order to study asymptotic stability, let us consider the equation (9.39) with $\mathbf{g}^{(k+1)} = \mathbf{0}\ \forall k \geq 0$.

If $\theta = 0$, it follows that

$$\mathbf{u}^k = (\mathrm{I} - \mu\Delta t\mathrm{A}/h^2)^k\mathbf{u}^0, \qquad k = 1, 2, \ldots$$

whence $\mathbf{u}^k \to \mathbf{0}$ as $k \to \infty$ iff

$$\rho(\mathrm{I} - \mu\Delta t\mathrm{A}/h^2) < 1. \tag{9.40}$$

On the other hand, the eigenvalues λ_j of A are given by

$$\lambda_j = 2 - 2\cos(j\pi/(N+1)) = 4\sin^2(j\pi/(2(N+1))), \quad j = 1, \ldots, N$$

(see Exercise 9.2). Then (9.40) is satisfied if

$$\Delta t < \frac{1}{2\mu}h^2.$$

As expected, the forward Euler method is conditionally asymptotically stable, under the condition that the time-step Δt should decay as the square of the grid spacing h.

In the case of the backward Euler method ($\theta = 1$), we would have from (9.39)

$$\mathbf{u}^k = \left[(\mathrm{I} + \mu\Delta t\mathrm{A}/h^2)^{-1}\right]^k \mathbf{u}^0, \qquad k = 1, 2, \ldots$$

Since all the eigenvalues of the matrix $(\mathrm{I}+\mu\Delta t\mathrm{A}/h^2)^{-1}$ are real, positive and strictly less than 1 for every value of Δt, this scheme is unconditionally asymptotically stable. More generally, the θ-scheme is unconditionally asymptotically stable for all the values $1/2 \le \theta \le 1$, and conditionally asymptotically stable if $0 \le \theta < 1/2$ (see, for instance, [QSS07, Chapter 13]).

As far as the accuracy of the θ-method is concerned, its local truncation error behaves like $\Delta t + h^2$ if $\theta \neq \frac{1}{2}$ while it is of the order of $\Delta t^2 + h^2$ if $\theta = \frac{1}{2}$. The latter is the *Crank-Nicolson method* (see Section 8.4) and is therefore unconditionally asymptotically stable; the corresponding global (in both space and time) discretization scheme is second-order accurate with respect to both Δt and h.

The same conclusions hold for the heat equation in a two-dimensional domain. In this case in the scheme (9.38) one must substitute to the matrix A/h^2 the finite difference matrix defined in (9.33).

Program 9.3 solves numerically the heat equation on the time interval $(0, T)$ and on the domain $\Omega = (a, b)$ using the θ-method. The input parameters are the vectors `xspan=[a,b]` and `tspan=[0,T]`, the number of discretization intervals in space (`nstep(1)`) and in time (`nstep(2)`), the scalar `mu` which contains the positive real coefficient μ, the function handles `u0`, `fun` and `g` associated with the initial function $u^0(x)$, the right hand side $f(x, t)$ and the Dirichlet datum $g(x, t)$, respectively. Finally, the variable `theta` contains the coefficient θ. The output variable `uh` contains the numerical solution at the final time $t = T$.

Program 9.3. heattheta: θ-method for the one-dimensional heat equation

```
function [xh,uh]=heattheta(xspan,tspan,nstep,mu,...
               u0,g,f,theta,varargin)
%HEATTHETA Solves the heat equation with the
% theta-method.
% [XH,UH]=HEATTHETA(XSPAN,TSPAN,NSTEP,MU,U0,G,F,THETA)
% solves the heat equation D U/DT - MU D^2U/DX^2 = F
% in (XSPAN(1),XSPAN(2)) X (TSPAN(1),TSPAN(2)) using
% the theta-method with initial condition U(X,0)=U0(X)
% and Dirichlet boundary conditions U(X,T)=G(X,T) at
% X=XSPAN(1) and X=XSPAN(2).
% MU is a positive constant, F=F(X,T), G=G(X,T) and
% U0=U0(X) are function handles.
% NSTEP(1) is the number of space integration intervals
% NSTEP(2) is the number of time-integration intervals
% XH contains the nodes of the discretization.
% UH contains the numerical solutions at time TSPAN(2).
% [XH,UH]=HEATTHETA(XSPAN,TSPAN,NSTEP,MU,U0,G,F,...
% THETA,P1,P2,...) passes the additional parameters
% P1,P2,...to the functions U0,G,F.

h  = (xspan(2)-xspan(1))/nstep(1);
dt = (tspan(2)-tspan(1))/nstep(2);
N = nstep(1)+1;
e = ones(N,1);
D = spdiags([-e 2*e -e],[-1,0,1],N,N);
I = speye(N);
A = I+mu*dt*theta*D/h^2;
An = I-mu*dt*(1-theta)*D/h^2;
A(1,:) = 0; A(1,1) = 1;
A(N,:) = 0; A(N,N) = 1;
xh = (linspace(xspan(1),xspan(2),N))';
fn = f(xh,tspan(1),varargin{:});
un = u0(xh,varargin{:});
[L,U]=lu(A);
for t = tspan(1)+dt:dt:tspan(2)
    fn1 = f(xh,t,varargin{:});
    rhs = An*un+dt*(theta*fn1+(1-theta)*fn);
    temp = g([xspan(1),xspan(2)],t,varargin{:});
    rhs([1,N]) = temp;
    uh = L\rhs; uh = U\uh; fn = fn1; un = uh;
end
return
```

Example 9.3 We consider the heat equation (9.4) in $(a,b) = (0,1)$ with $\mu = 1$, $f(x,t) = -\sin(x)\sin(t)+\sin(x)\cos(t)$, initial condition $u(x,0) = \sin(x)$ and boundary conditions $u(0,t) = 0$ and $u(1,t) = \sin(1)\cos(t)$. In this case the exact solution is $u(x,t) = \sin(x)\cos(t)$. In Figure 9.9 we compare the behavior of the errors $\max_{i=0,\ldots,N} |u(x_i,1) - u_i^M|$ with respect to the time-step on a uniform grid in space with $h = 0.002$. $\{u_i^M\}$ are the values of the finite difference solution computed at time $t^M = 1$. As expected, for $\theta = 0.5$ the θ-method is second order accurate until when the time-step is so small that the spatial error dominates over the error due to the temporal discretization. ■

Example 9.4 (Thermodynamics) We consider a homogeneous, three meters long aluminium bar with uniform section. We are interested in simulating the evolution of the temperature in the bar starting from a suitable initial condition, by solving the heat equation (9.5). If we impose adiabatic conditions on the lateral surface of the bar (i.e. homogeneous Neumann conditions), and Dirichlet conditions at the end sections of the bar, the temperature only depends on the axial space variable (denoted by x). Thus the problem can be modeled by the one-dimensional heat equation (9.7) with $f = 0$, completed by the initial condition at $t = t_0$ and by Dirichlet boundary conditions at the endpoints of the reduced computational domain $\Omega = (0,L)$

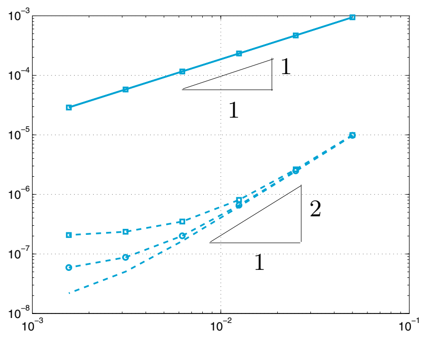


Figure 9.9. Error versus Δt for the θ-method (for $\theta = 1$, *solid line*, and $\theta = 0.5$ *dashed line*), for three different values of h: 0.008 (□), 0.004 (○) and 0.002 (*no symbols*)

($L = 3$m). Pure aluminium has thermal conductivity $k = 237$ W/(m K), density $\rho = 2700$kg/m^3 and specific heat capacity $c = 897$ J/(kg K), then its thermal diffusivity is $\mu = 9.786 \cdot 10^{-5}$m^2/s. Finally we consider the initial condition $T(x, 0) = 500$ K if $x \in (1, 2)$, 250 K otherwise and the Dirichlet boundary conditions $T(0, t) = T(3, t) = 250$ K. In Figure 9.10 we report the evolution of the temperature starting from the initial data, computed by the backward Euler method ($\theta = 1$, *left*) and by the Crank-Nicolson method ($\theta = 0.5$, *right*) (using Program 9.3).
The results show that when the time-step is large ($\Delta t = 20$sec), the Crank-Nicolson method is unstable because of the low smoothness of the initial datum (about this point, see also [QV94, Chapter 11]). On the contrary, the implicit Euler method provides a stable solution because it is more dissipative than Crank-Nicolson. Both methods compute a solution that decays to the correct value 250 K as $t \to \infty$. ■

9.2.7 Finite element approximation of the one-dimensional heat equation

The space discretization of the heat equation (9.4) with homogeneous Dirichlet boundary conditions $u(a, t) = u(b, t) = 0$, $\forall t > 0$ can be accomplished using the Galerkin finite element method by proceeding as we did in Section 9.2.3 for the Poisson equation. First, for all $t > 0$ we multiply (9.4) by a test function $v = v(x) \in C^1([a, b])$ and we integrate the resulting equation over (a, b). For all $t > 0$ we therefore look for a function $t \to u(x, t) \in C^1([a, b])$ such that

$$\int_a^b \frac{\partial u}{\partial t}(x,t)v(x)\mathrm{d}x + \int_a^b \mu\frac{\partial u}{\partial x}(x,t)\frac{\mathrm{d}v}{\mathrm{d}x}(x)\mathrm{d}x = \tag{9.41}$$

$$= \int_a^b f(x)v(x)\mathrm{d}x \qquad \forall v \in C^1([a, b]),$$

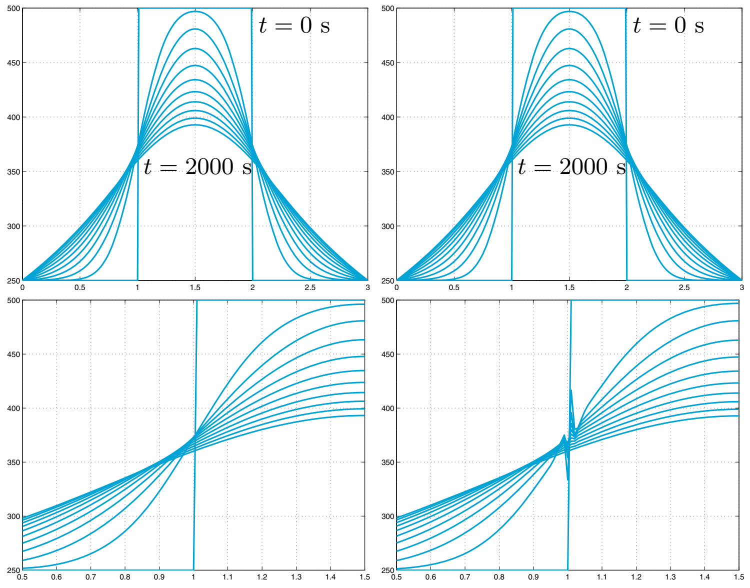


Figure 9.10. Temperature profiles in an aluminium bar at different time levels (from $t = 0$ to $t = 2000$ seconds with time-step Δt of 0.25 seconds (*top*) and 20 seconds (*bottom*)), obtained using the backward Euler method (*left*) and the Crank-Nicolson method (*right*). In both cases, the space discretization is carried out by centered finite differences with steplength $h = 0.01$. The zoom on the solutions for $\Delta t = 20$sec (at bottom) shows instability of the Crank-Nicolson scheme

with $u(x, 0) = u^0(x)$. To simplify notations, from now on the dependence on variable x in u, v and f will be understood.

Equation (9.41) keeps holding also for functions v less regular than $C^1([a, b])$, e.g., like those of the space V_h defined in (9.26). Therefore, we consider the following Galerkin formulation: $\forall t > 0$, find $u_h(t) \in V_h$ such that

$$\int_a^b \frac{\partial u_h}{\partial t}(t) v_h \mathrm{d}x + \int_a^b \mu \frac{\partial u_h}{\partial x}(t) \frac{\mathrm{d}v_h}{\mathrm{d}x} \mathrm{d}x = \int_a^b f(t) v_h \mathrm{d}x \quad \forall v_h \in V_h, \quad (9.42)$$

where $u_h(0) = u_h^0$ and $u_h^0 \in V_h$ is a convenient approximation of u^0. Formulation (9.42) is called *semi-discretization* of problem (9.41), since only the space discretization (not yet the time) was carried out.

For what concerns the finite element discretization of (9.42), let us consider the basis functions φ_j introduced in Section 9.2.3. Then, the solution u_h of (9.42) can be sought under the form

$$u_h(t) = \sum_{j=1}^{N} u_j(t) \varphi_j,$$

where $\{u_j(t)\}$ are the unknown coefficients and N is the dimension of V_h.
Then, from (9.42) we obtain

$$\int_a^b \sum_{j=1}^N \frac{\mathrm{d}u_j}{\mathrm{d}t}(t)\varphi_j\varphi_i \mathrm{d}x + \mu \int_a^b \sum_{j=1}^N u_j(t) \frac{\mathrm{d}\varphi_j}{\mathrm{d}x}\frac{\mathrm{d}\varphi_i}{\mathrm{d}x}\mathrm{d}x = \\ = \int_a^b f(t)\varphi_i \mathrm{d}x, \qquad i=1,\ldots,N$$

that is,

$$\sum_{j=1}^N \frac{\mathrm{d}u_j}{\mathrm{d}t}(t)\int_a^b \varphi_j\varphi_i \mathrm{d}x + \mu \sum_{j=1}^N u_j(t) \int_a^b \frac{\mathrm{d}\varphi_j}{\mathrm{d}x}\frac{\mathrm{d}\varphi_i}{\mathrm{d}x}\mathrm{d}x = \\ = \int_a^b f(t)\varphi_i \mathrm{d}x, \qquad i=1,\ldots,N.$$

Using the same notations as in (9.37) we obtain

$$\mathrm{M}\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t}(t) + \mathrm{A}_{\mathrm{fe}}\mathbf{u}(t) = \mathbf{f}_{\mathrm{fe}}(t), \tag{9.43}$$

where $(\mathrm{A}_{\mathrm{fe}})_{ij} = \mu\int_a^b \frac{\mathrm{d}\varphi_j}{\mathrm{d}x}\frac{\mathrm{d}\varphi_i}{\mathrm{d}x}\mathrm{d}x$, $(\mathbf{f}_{\mathrm{fe}}(t))_i = \int_a^b f(t)\varphi_i\mathrm{d}x$ and $\mathrm{M}_{ij} = (\int_a^b \varphi_j\varphi_i \mathrm{d}x)$ for $i,j=1,\ldots,N$. M is called the *mass matrix*. Since it is not singular, the system of ordinary differential equations (9.43) can be written in normal form as

$$\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t}(t) = -\mathrm{M}^{-1}\mathrm{A}_{\mathrm{fe}}\mathbf{u}(t) + \mathrm{M}^{-1}\mathbf{f}_{\mathrm{fe}}(t). \tag{9.44}$$

To solve (9.43) approximately, we can still apply the θ-method and obtain

$$\mathrm{M}\frac{\mathbf{u}^{k+1}-\mathbf{u}^k}{\Delta t} + \mathrm{A}_{\mathrm{fe}}\left[\theta\mathbf{u}^{k+1} + (1-\theta)\mathbf{u}^k\right] = \theta\mathbf{f}_{\mathrm{fe}}^{k+1} + (1-\theta)\mathbf{f}_{\mathrm{fe}}^k. \tag{9.45}$$

As usual, the upper index k means that the quantity at hand is computed at time $t^k = k\Delta t$, $\Delta t > 0$ being the time discretization step. As in the finite difference case, for $\theta = 0, 1$ and $1/2$, we respectively obtain the forward Euler, backward Euler and Crank-Nicolson methods, the latter being the only one which is second-order accurate with respect to Δt.
For each k, (9.45) is a linear system whose matrix is

$$\mathrm{K} = \frac{1}{\Delta t}\mathrm{M} + \theta\mathrm{A}_{\mathrm{fe}}.$$

Since both matrices M and A_{fe} are symmetric and positive definite, the matrix K is also symmetric and positive definite. Moreover, K is independent of k and then it can be factorized once at $t = 0$. For the one-dimensional case that we are handling, this factorization is based on the Thomas method (see Section 5.6) and it requires a number of operation proportional to N. In the multidimensional case the use of the Cholesky factorization $\mathrm{K} = \mathrm{R}^T\mathrm{R}$, R being an upper triangular matrix (see (5.17)), will be more convenient. Consequently, at each time level the following two linear triangular systems, each of size equal to N, must be solved:

$$\begin{cases} \mathrm{R}^T\mathbf{y} = \left[\frac{1}{\Delta t}\mathrm{M} - (1-\theta)\mathrm{A}_{\mathrm{fe}}\right]\mathbf{u}^k + \theta\mathbf{f}_{\mathrm{fe}}^{k+1} + (1-\theta)\mathbf{f}_{\mathrm{fe}}^k, \\ \mathrm{R}\mathbf{u}^{k+1} = \mathbf{y}. \end{cases}$$

When $\theta = 0$, a suitable diagonalization of M would allow to decouple the system equations (9.45). The procedure is carried out by the so-called *mass-lumping* in which we approximate M by a non-singular diagonal matrix $\widetilde{\mathrm{M}}$. In the case of piecewise linear finite elements, $\widetilde{\mathrm{M}}$ can be obtained using the composite trapezoidal formula over the nodes $\{x_i\}$ to evaluate the integrals $\int_a^b \varphi_j\varphi_i\,\mathrm{d}x$, obtaining $\tilde{m}_{ij} = h\delta_{ij}$, $i,j = 1,\ldots,N$.

If $\theta \geq 1/2$, the θ-method is unconditionally stable for every positive value of Δt, while if $0 \leq \theta < 1/2$ the θ-method is stable only if

$$0 < \Delta t \leq \frac{2}{(1-2\theta)\lambda_{\max}(\mathrm{M}^{-1}\mathrm{A}_{\mathrm{fe}})},$$

to this aim see [Qua13, Chap. 5]. Moreover, it is possible to prove that there exist two positive constants c_1 and c_2, independent of h, such that

$$c_1 h^{-2} \leq \lambda_{\max}(\mathrm{M}^{-1}\mathrm{A}_{\mathrm{fe}}) \leq c_2 h^{-2}$$

(see [QV94, Section 6.3.2] for a proof). Thanks to this property, if $0 \leq \theta < 1/2$ the method is stable only if

$$0 < \Delta t \leq C_1(\theta)h^2, \tag{9.46}$$

where $C_1(\theta)$ is a suitable constant independent of both discretization parameters h and Δt.

9.3 Hyperbolic equations: a scalar pure advection problem

Let us consider the following scalar hyperbolic problem

$$\begin{cases} \dfrac{\partial u}{\partial t} + a\dfrac{\partial u}{\partial x} = 0, & x \in \mathbb{R},\, t > 0, \\ u(x,0) = u^0(x), & x \in \mathbb{R}, \end{cases} \tag{9.47}$$

where a is a positive real number. Its solution is given by

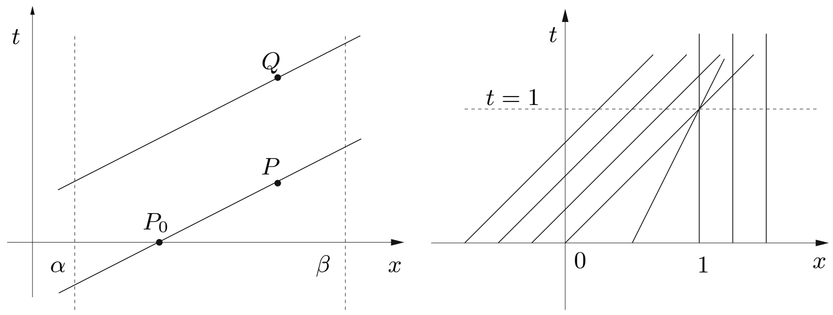


Figure 9.11. At left: examples of characteristics which are straight lines issuing from the points P and Q. At right: characteristic straight lines for the Burgers equation (9.51)

$$u(x,t) = u^0(x - at),\ t \geq 0,$$

and represents a wave travelling with velocity a. The curves $(x(t), t)$ in the plain (x, t), that satisfy the following scalar ordinary differential equation

$$\begin{cases} \dfrac{dx}{dt}(t) = a, & t > 0, \\ x(0) = x_0, \end{cases} \tag{9.48}$$

are called *characteristic curves* (or, simply, *characteristics*), and are the straight lines $x(t) = x_0 + at$, $t > 0$. The solution of (9.47) remains constant along them since

$$\frac{du}{dt} = \frac{\partial u}{\partial t} + \frac{\partial u}{\partial x}\frac{dx}{dt} = 0 \qquad \text{on} \quad (x(t), t).$$

For the more general problem

$$\begin{cases} \dfrac{\partial u}{\partial t} + a\dfrac{\partial u}{\partial x} + a_0 u = f, & x \in \mathbb{R}, \quad t > 0, \\ u(x,0) = u^0(x), & x \in \mathbb{R}, \end{cases} \tag{9.49}$$

where a, a_0 and f are given functions of the variables (x, t), the characteristic curves are still defined as in (9.48). In this case, the solutions of (9.49) satisfy along the characteristics the following differential equation

$$\frac{du}{dt} = f - a_0 u \qquad \text{on } (x(t), t).$$

Let us now consider problem (9.47) on a bounded interval $[\alpha, \beta]$

$$\begin{cases} \dfrac{\partial u}{\partial t} + a\dfrac{\partial u}{\partial x} = 0, & x \in (\alpha, \beta),\ t > 0, \\ u(x,0) = u^0(x), & x \in (\alpha, \beta). \end{cases} \tag{9.50}$$

Let us start with $a > 0$. Since u is constant along the characteristics, from Figure 9.11, left, we deduce that the value of the solution at P attains the value of u^0 at P_0, the foot of the characteristic issuing from P. On the other hand, the characteristic issuing from Q intersects the straight line $x(t) = \alpha$ at a certain time $t = \bar{t} > 0$. Thus, the point $x = \alpha$ is an *inflow* point and it is necessary to assign there a boundary value for u, for every $t > 0$. Notice that if $a < 0$ then the inflow point is $x = \beta$ and it is necessary to assign there a boundary value for u, for every $t > 0$.

Referring to problem (9.47) it is worth noting that if u^0 is discontinuous at a point x_0, then such a discontinuity propagates along the characteristics issuing from x_0. This process can be made rigorous by introducing the concept of *weak solutions* of hyperbolic problems, see e.g. [GR96]. Another reason for introducing weak solutions is that in the case of nonlinear hyperbolic problems the characteristic lines can intersect: in this case the solution cannot be continuous and no classical solution does exist.

Example 9.5 (Burgers equation) Let us consider the Burgers equation

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = 0, \qquad x \in \mathbb{R}, \quad t > 0, \tag{9.51}$$

which is perhaps the simplest nontrivial example of a nonlinear hyperbolic equation. Taking as initial condition

$$u(x,0) = u^0(x) = \begin{cases} 1, & x \le 0, \\ 1 - x, & 0 < x \le 1, \\ 0, & x > 1, \end{cases}$$

the characteristic line issuing from the point $(x_0, 0)$ is given by

$$x(t) = x_0 + tu^0(x_0) = \begin{cases} x_0 + t, & x_0 \le 0, \\ x_0 + t(1 - x_0), & 0 < x_0 \le 1, \\ x_0, & x_0 > 1. \end{cases}$$

Notice that the characteristic lines do not intersect only if $t < 1$ (see Figure 9.11, right). ■

9.3.1 Finite difference discretization of the scalar transport equation

The half-plane $\{(x,t) :\ -\infty < x < \infty,\ t > 0\}$ is discretized by choosing a spatial grid size $\Delta x > 0$ (the parameter named h until now), a temporal step $\Delta t > 0$ and the grid points (x_j, t^n) as follows

$$x_j = j\Delta x, \qquad j \in \mathbb{Z}, \qquad t^n = n\Delta t, \qquad n \in \mathbb{N}.$$

Let us set

$$\lambda = \Delta t/\Delta x,$$

and define $x_{j+1/2} = x_j + \Delta x/2$. We look for discrete solutions u_j^n which approximate the values $u(x_j, t^n)$ of the exact solution for any j, n. Quite often, explicit methods are employed for advancing in time hyperbolic initial-value problems.
Any explicit finite-difference method can be written in the form

$$u_j^{n+1} = u_j^n - \lambda(h_{j+1/2}^n - h_{j-1/2}^n), \tag{9.52}$$

where $h_{j+1/2}^n = h(u_j^n, u_{j+1}^n)$ for every j and $h(\cdot,\cdot)$ is a function, to be properly chosen, that is called the *numerical flux*.

In what follows we will illustrate several instances of explicit methods for the approximation of problem (9.47):

1. *forward Euler/centered*

$$u_j^{n+1} = u_j^n - \frac{\lambda}{2}a(u_{j+1}^n - u_{j-1}^n), \tag{9.53}$$

which can be cast in the form (9.52) by setting

$$h_{j+1/2}^n = \frac{1}{2}a(u_{j+1}^n + u_j^n); \tag{9.54}$$

2. *Lax-Friedrichs*

$$u_j^{n+1} = \frac{1}{2}(u_{j+1}^n + u_{j-1}^n) - \frac{\lambda}{2}a(u_{j+1}^n - u_{j-1}^n), \tag{9.55}$$

which is of the form (9.52) with

$$h_{j+1/2}^n = \frac{1}{2}[a(u_{j+1}^n + u_j^n) - \lambda^{-1}(u_{j+1}^n - u_j^n)]; \tag{9.56}$$

3. *Lax-Wendroff*

$$u_j^{n+1} = u_j^n - \frac{\lambda}{2}a(u_{j+1}^n - u_{j-1}^n) + \frac{\lambda^2}{2}a^2(u_{j+1}^n - 2u_j^n + u_{j-1}^n), \tag{9.57}$$

which can be written in the form (9.52) provided that

$$h_{j+1/2}^n = \frac{1}{2}[a(u_{j+1}^n + u_j^n) - \lambda a^2(u_{j+1}^n - u_j^n)]; \tag{9.58}$$

4. *Upwind (or forward Euler/decentered)*

$$u_j^{n+1} = u_j^n - \frac{\lambda}{2}a(u_{j+1}^n - u_{j-1}^n) + \frac{\lambda}{2}|a|(u_{j+1}^n - 2u_j^n + u_{j-1}^n), \tag{9.59}$$

which fits the form (9.52) when the numerical flux is defined to be

$$h_{j+1/2}^n = \frac{1}{2}[a(u_{j+1}^n + u_j^n) - |a|(u_{j+1}^n - u_j^n)]. \tag{9.60}$$

Table 9.1. Artificial viscosity, artificial diffusion flux, and truncation error for Lax-Friedrichs, Lax-Wendroff and upwind methods

method	k	$h^{diff}_{j+1/2}$	$\tau(\Delta t, \Delta x)$
Lax-Friedrichs	Δx^2	$-\dfrac{1}{2\lambda}(u_{j+1}-u_j)$	$\mathcal{O}\left(\Delta x^2/\Delta t + \Delta t + \Delta x^2\right)$
Lax-Wendroff	$a^2\Delta t^2$	$-\dfrac{\lambda a^2}{2}(u_{j+1}-u_j)$	$\mathcal{O}\left(\Delta t^2 + \Delta x^2 + \Delta t\Delta x^2\right)$
upwind	$\lvert a\rvert\Delta x\Delta t$	$-\dfrac{\lvert a\rvert}{2}(u_{j+1}-u_j)$	$\mathcal{O}(\Delta t + \Delta x)$

Each one of the last three methods can be obtained from the forward Euler/centered method by adding a term proportional to the centered finite difference (4.9), so that they can be written in the equivalent form

$$u_j^{n+1} = u_j^n - \frac{\lambda}{2}a(u_{j+1}^n - u_{j-1}^n) + \frac{1}{2}k\frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{(\Delta x)^2}. \tag{9.61}$$

The last term represents indeed a discretization of the second-order derivative

$$\frac{k}{2}\frac{\partial^2 u}{\partial x^2}(x_j, t^n).$$

The coefficient $k > 0$ plays the role of artificial viscosity. Its expression is given for the three previous cases in Table 9.1. Consequently, the numerical flux for each scheme can be equivalently written as

$$h_{j+1/2} = h^{FE}_{j+1/2} + h^{diff}_{j+1/2},$$

where $h^{FE}_{j+1/2}$ is the numerical flux of the forward Euler/centered scheme (which is given in (9.54)) and the *artificial diffusion flux* $h^{diff}_{j+1/2}$ for the three cases is also reported in Table 9.1.
The most classical implicit method is the *backward Euler/centered* scheme

$$u_j^{n+1} + \frac{\lambda}{2}a(u_{j+1}^{n+1} - u_{j-1}^{n+1}) = u_j^n. \tag{9.62}$$

It can still be written in the form (9.52) provided that h^n is replaced by h^{n+1}. In the example at hand, the numerical flux is the same as for the forward Euler/centered method.

9.3.2 Finite difference analysis for the scalar transport equation

The convergence analysis of finite difference methods introduced in the previous Section requires that both consistency and stability hold.
Consider for instance, the forward Euler/centered method (9.53). As

done in Section 8.3.1, denoting by u the exact solution of problem (9.47), the *local truncation error* at (x_j, t^n) represents, up to a factor $1/\Delta t$, the error that would be generated by forcing the exact solution to satisfy that specific numerical scheme. In particular for the forward Euler/centered method it is defined as follows

$$\tau_j^n = \frac{u(x_j, t^{n+1}) - u(x_j, t^n)}{\Delta t} + a\frac{u(x_{j+1}, t^n) - u(x_{j-1}, t^n)}{2\Delta x},$$

while the *(global) truncation error* is defined as

$$\tau(\Delta t, \Delta x) = \max_{j,n} |\tau_j^n|.$$

When $\tau(\Delta t, \Delta x)$ goes to zero as Δt and Δx tend to zero independently, the numerical scheme is said to be *consistent*.

More in general, we say that a numerical method is of *order* p in time and of *order* q in space (for suitable positive values p and q) if, for a sufficiently smooth solution of the exact problem,

$$\tau(\Delta t, \Delta x) = \mathcal{O}(\Delta t^p + \Delta x^q).$$

Finally, we say that a numerical scheme is *convergent* (in the maximum norm) if

$$\lim_{\Delta t, \Delta x \to 0} \max_{j,n} |u(x_j, t^n) - u_j^n| = 0.$$

If the exact solution is regular enough, using Taylor's expansion conveniently, we can characterize the truncation error of the methods previously introduced. For the forward (or backward) Euler/centered method it is $\mathcal{O}(\Delta t + \Delta x^2)$. For the other methods, see Table 9.1.

As of stability, we say that a numerical scheme for the approximation of a hyperbolic (either linear or nonlinear) problem is *stable* if, for any time T, there exist two constants $C_T > 0$ (possibily depending on T) and $\delta_0 > 0$, such that

$$\|\mathbf{u}^n\|_\Delta \leq C_T \|\mathbf{u}^0\|_\Delta, \tag{9.63}$$

for any n such that $n\Delta t \leq T$ and for any Δt, Δx such that $0 < \Delta t \leq \delta_0$, $0 < \Delta x \leq \delta_0$. The symbol $\|\cdot\|_\Delta$ stands for a suitable discrete norm, there are three instances:

$$\|\mathbf{v}\|_{\Delta,p} = \left(\Delta x \sum_{j=-\infty}^{\infty} |v_j|^p\right)^{\frac{1}{p}} \quad \text{for } p = 1, 2, \quad \|\mathbf{v}\|_{\Delta,\infty} = \sup_j |v_j|. \tag{9.64}$$

Courant, Friedrichs and Lewy [CFL28] have proved that a necessary and sufficient condition for any explicit scheme of the form (9.52) to be stable is that the time and space discretization steps must obey the following condition

$$|a\lambda| \leq 1, \text{ i.e. } \Delta t \leq \frac{\Delta x}{|a|} \tag{9.65}$$

which is known as the *CFL condition*. The adimensional number $a\lambda$ (a is a velocity) is commonly referred to as the *CFL number*. If a is not constant the CFL condition becomes

$$\Delta t \leq \frac{\Delta x}{\sup\limits_{x\in\mathbb{R},\ t>0} |a(x,t)|}.$$

It is possible to prove that

1. the *forward Euler/centered* method (9.53) is unconditionally unstable, i.e. it is unstable for any possible choice of $\Delta x > 0$ and $\Delta t > 0$;
2. the *upwind* method (also called *forward Euler/decentered* method) (9.59) is conditionally stable with respect to the $\|\cdot\|_{\Delta,1}$ norm, i.e.

   $$\|\mathbf{u}^n\|_{\Delta,1} \leq \|\mathbf{u}^0\|_{\Delta,1} \qquad \forall n \geq 0,$$

   provided that the CFL condition (9.65) is satisfied; the same result can be proved also for both *Lax-Friedrichs* (9.55) and *Lax-Wendroff* (9.57) schemes;
3. the *backward Euler/centered* method (9.62) is unconditionally stable with respect to the $\|\cdot\|_{\Delta,2}$ norm, i.e., for any $\Delta t > 0$

   $$\|\mathbf{u}^n\|_{\Delta,2} \leq \|\mathbf{u}^0\|_{\Delta,2} \qquad \forall n \geq 0.$$

   See Exercise 9.11.

For a proof of the these results see, e.g., [QSS07, Chap. 13] and [Qua13, Chap. 12].

We want now to mention two important features of a numerical scheme: *dissipation* and *dispersion*. To this aim, let us suppose that the initial datum $u^0(x)$ of problem (9.47) is $2\pi-$periodic so that it can be expanded in a Fourier series as

$$u^0(x) = \sum_{k=-\infty}^{\infty} \alpha_k e^{ikx},$$

where

$$\alpha_k = \frac{1}{2\pi}\int_0^{2\pi} u^0(x)e^{-ikx}\mathrm{d}x$$

is the $k-$th Fourier coefficient of $u^0(x)$. The exact solution u of problem (9.47) satisfies (formally) the nodal conditions

$$u(x_j, t^n) = \sum_{k=-\infty}^{\infty} \alpha_k e^{ikj\Delta x} (g_k)^n, \quad j \in \mathbb{Z}, n \in \mathbb{N} \tag{9.66}$$

with $g_k = e^{-iak\Delta t}$, while the numerical solution u_j^n, computed by one of the schemes introduced in Section 9.3.1, reads

$$u_j^n = \sum_{k=-\infty}^{\infty} \alpha_k e^{ikj\Delta x} (\gamma_k)^n, \quad j \in \mathbb{Z}, \quad n \in \mathbb{N}. \tag{9.67}$$

The form of coefficients $\gamma_k \in \mathbb{C}$ depends on the particular numerical scheme used; for instance, for the scheme (9.53) we can show that $\gamma_k = 1 - a\lambda i \sin(k\Delta x)$.
We notice that, while $|g_k| = 1$ for any $k \in \mathbb{Z}$, the values $|\gamma_k|$ depend on the *CFL number* $a\lambda$, and then also on the chosen discretization. Precisely, by choosing $\|\cdot\|_\Delta = \|\cdot\|_{\Delta,2}$, one can prove that a necessary and sufficient condition for a given numerical scheme to satisfy the stability inequality (9.63) is that $|\gamma_k| \le 1$, $\forall k \in \mathbb{Z}$. The ratio $\epsilon_a(k) = |\gamma_k|/|g_k| = |\gamma_k|$ is the so-called *dissipation coefficient* (or *amplification coefficient*) of the $k-$th harmonic associated with the numerical scheme. We recall that the exact solution of (9.47) is the travelling wave $u(x,t) = u^0(x - at)$ whose amplitude is independent of time; as of its numerical approximation (9.67), the smaller $\epsilon_a(k)$, the higher the reduction of the wave amplitude and, whence the higher the numerical dissipation. Moreover, if the stability condition is violated, then the wave amplitude will increase and a *blow-up* of the numerical solution will occur at sufficiently large times.

Besides dissipation, numerical schemes introduce also dispersion, that is either a delay or an advance in the wave propagation. To understand this phenomenon we write g_k and γ_k as follows:

$$g_k = e^{-ia\lambda\phi_k}, \qquad \gamma_k = |\gamma_k| e^{-i\omega\Delta t} = |\gamma_k| e^{-i\frac{\omega}{k}\lambda\phi_k},$$

$\phi_k = k\Delta x$ being the so-called *phase angle* associated to the $k-$th harmonic.
By comparing g_k with γ_k and recalling that a is the propagation velocity of the "exact" wave, we define *dispersion coefficient* associated to the kth harmonic the value $\epsilon_d(k) = \frac{\omega}{ak} = \frac{\omega\Delta t}{\phi_k a\lambda}$.

In Figures 9.12 and 9.13 we report the exact solution of problem (9.50) (for $a = 1$) and the numerical solutions obtained by some of the schemes presented in Section 9.3.1. The initial datum is

$$u^0(x) = \begin{cases} \sin(2\pi x/\ell) & -1 \le x \le \ell \\ 0 & \ell < x < 3, \end{cases} \tag{9.68}$$

of wavelength $\ell = 1$ (left) and $\ell = 1/2$ (right). In both cases the CFL number is equal to 0.8. For $\ell = 1$ we have chosen $\Delta x = \ell/20 = 1/20$, so

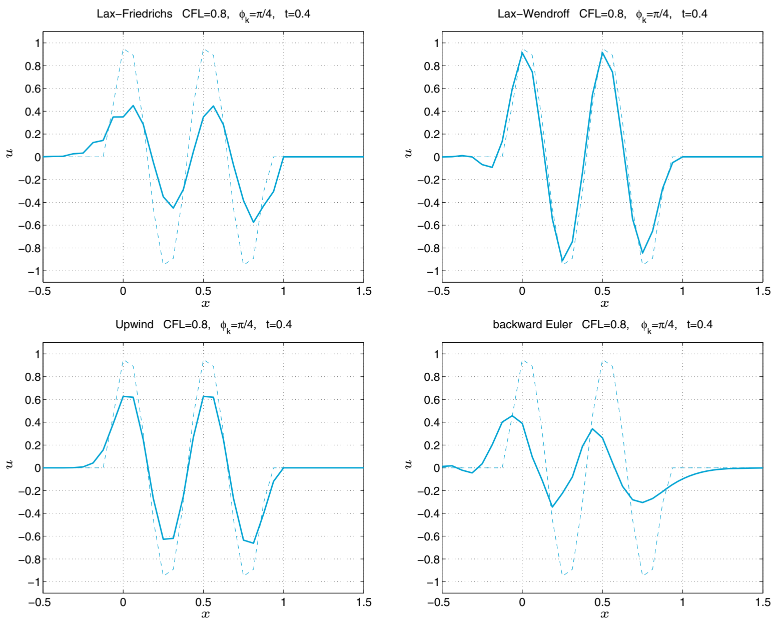


Figure 9.12. Exact solution (*dashed line*) and numerical solution (*solid line*) of problem (9.50) at $t = 0.4$, with $a = 1$ and initial datum given by (9.68) with equal wavelength $\ell = 1/2$

that $\phi_k = 2\pi\Delta x/\ell = \pi/10$ and $\Delta t = 1/25$. For $\ell = 1/2$ we have chosen $\Delta x = \ell/8 = 1/16$, so that $\phi_k = \pi/4$ and $\Delta t = 1/20$.

In Figures 9.14 and 9.15 we display the dissipation and dispersion coefficients, respectively, versus the CFL number (at top) and the phase angle $\phi_k = k\Delta x$ (at bottom).
Notice from Figure 9.14 that, when CFL=0.8, the Lax-Wendroff scheme is the least dissipative one, this information is confirmed by the numerical solutions shown in Figure 9.13, for both $\phi_k = \pi/10$ and $\phi_k = \pi/4$. About the dispersion error, still for CFL=0.8, from Figure 9.15 it emerges that the upwind scheme features the lowest dispersion and shows a light phase advance; the Lax-Friederichs scheme has a considerable phase advance, while both Lax-Wendroff and implicit Euler/centered schemes show a phase delay. These conclusions are confirmed by the numerical solution shown in Figure 9.12.

Notice that the dissipation coefficient is responsible for the damping of the wave amplitude, while the dispersion coefficient is responsible for the inexact propagation velocity.

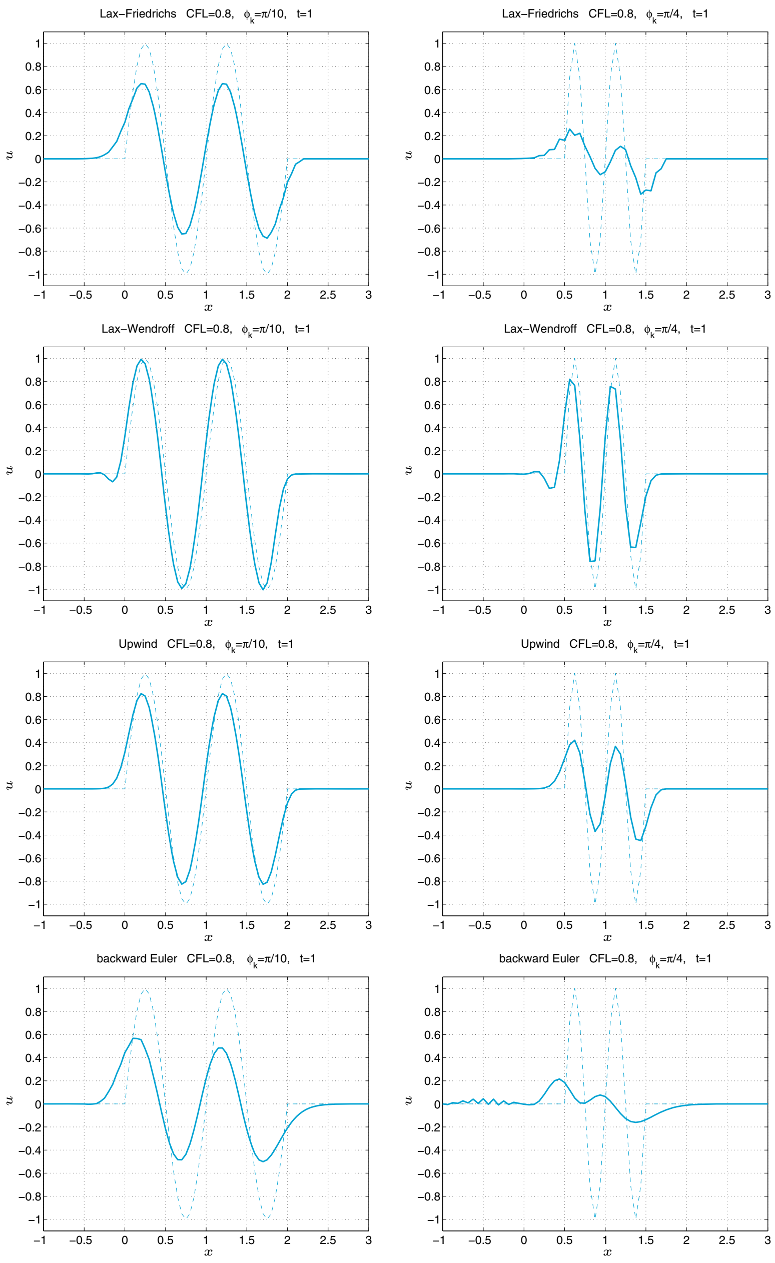


Figure 9.13. Exact solution (*dashed line*) and numerical solution (*solid line*) at $t = 1$ of problem (9.50) with $a = 1$ and initial datum given by (9.68) with wavelength $\ell = 1$ (left) and $\ell = 1/2$ (right)

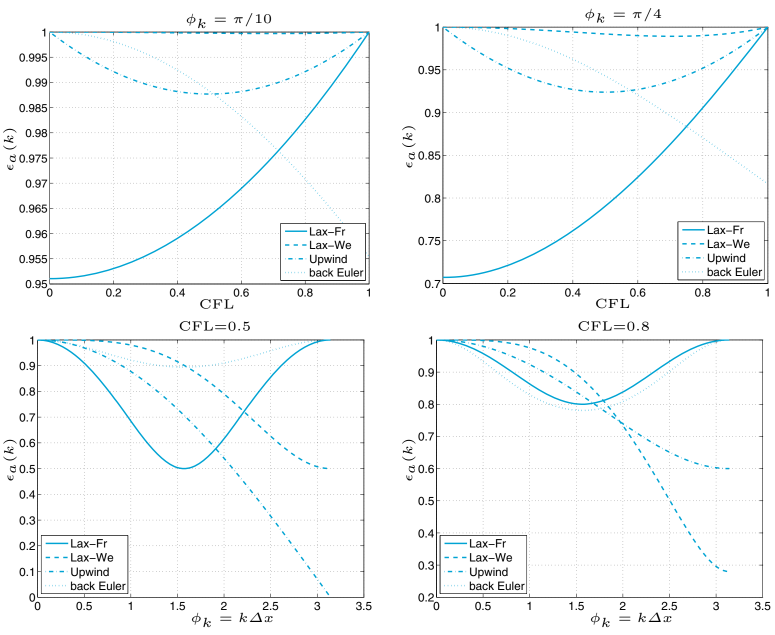


Figure 9.14. Dissipation coefficients

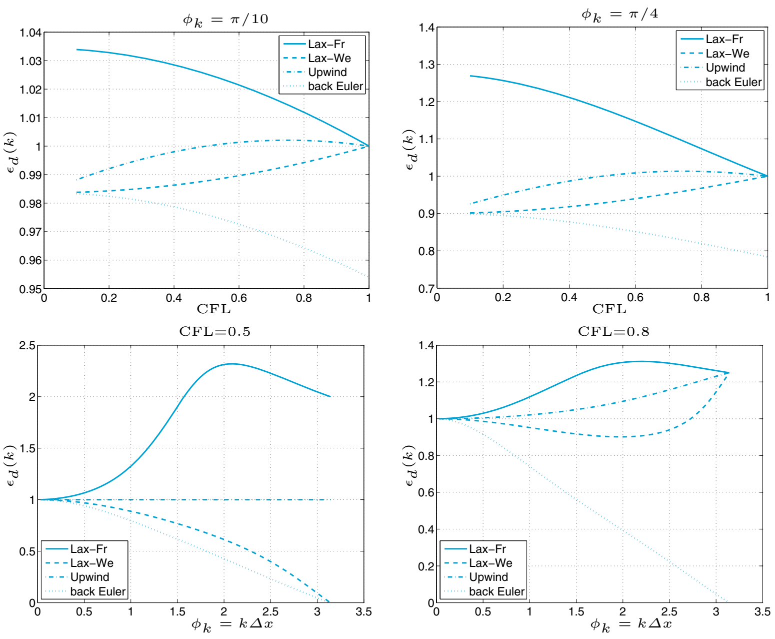


Figure 9.15. Dispersion coefficients

9.3.3 Finite element space discretization of the scalar advection equation

Following Section 9.2.3, a Galerkin semi-discrete approximation of problem (9.47) can be introduced as follows. Let us assume that $a = a(x) > 0$ $\forall x \in [\alpha, \beta]$, so that the node $x = \alpha$ coincides with the *inflow boundary.* For any $t > 0$, we complete system (9.47) with the boundary condition

$$u(\alpha, t) = \varphi(t), \qquad t > 0, \tag{9.69}$$

where φ is a given function of t.

After defining the space

$$V_h^{in} = \{v_h \in V_h : \ v_h(\alpha) = 0\},$$

we consider the following finite element approximation of problem (9.47), (9.69): for any $t \in (0, T)$ find $u_h(t) \in V_h$ such that

$$\begin{cases} \displaystyle\int_\alpha^\beta \frac{\partial u_h(t)}{\partial t} v_h \ \mathrm{d}x + \int_\alpha^\beta a \frac{\partial u_h(t)}{\partial x} v_h \ \mathrm{d}x = 0 & \forall \ v_h \in V_h^{in}, \\ u_h(t) = \varphi(t) & \text{at } x = \alpha, \end{cases} \tag{9.70}$$

with $u_h(0) = u_h^0 \in V_h$ being a suitable finite element approximation of the initial datum u^0, e.g. its piecewise polynomial interpolant.

The time discretization of (9.70) can be accomplished still by using finite difference schemes. If, for instance, we use the backward Euler method, for any $n \geq 0$, we have: find $u_h^{n+1} \in V_h$ such that

$$\frac{1}{\Delta t}\int_\alpha^\beta (u_h^{n+1} - u_h^n) v_h \ \mathrm{d}x + \int_\alpha^\beta a \frac{\partial u_h^{n+1}}{\partial x} v_h \ \mathrm{d}x = 0 \quad \forall v_h \in V_h^{in}, \tag{9.71}$$

with $u_h^{n+1}(\alpha) = \varphi^{n+1}$.

If $\varphi = 0$, we can conclude that

$$\|u_h^n\|_{\mathrm{L}^2(\alpha,\beta)} \leq \|u_h^0\|_{\mathrm{L}^2(\alpha,\beta)} \quad \forall n \geq 0,$$

which means that the backward Euler scheme is unconditionally stable with respect to the norm $\|v\|_{\mathrm{L}^2(\alpha,\beta)} = \left(\int_\alpha^\beta v^2(x)\mathrm{d}x\right)^{1/2}$.

See Exercises 9.10-9.14.

9.4 The wave equation

We consider now the following second-order hyperbolic equation in one dimension

$$\boxed{\frac{\partial^2 u}{\partial t^2} - c\frac{\partial^2 u}{\partial x^2} = f} \tag{9.72}$$

where c is a given positive constant.
When $f = 0$, the general solution of (9.72) is the so-called d'Alembert travelling-wave

$$u(x,t) = \psi_1(\sqrt{c}t - x) + \psi_2(\sqrt{c}t + x), \tag{9.73}$$

for arbitrary functions ψ_1 and ψ_2.

In what follows we consider problem (9.72) for $x \in (a,b)$ and $t > 0$, therefore we need to complete the differential equation with the initial data

$$u(x,0) = u_0(x) \text{ and } \frac{\partial u}{\partial t}(x,0) = v_0(x), \; x \in (a,b), \tag{9.74}$$

and the boundary data

$$u(a,t) = 0 \text{ and } u(b,t) = 0, \; t > 0. \tag{9.75}$$

In this case, u may represent the transverse displacement of an elastic vibrating string of length $b-a$, fixed at the endpoints, and c is a positive coefficient depending on the specific mass of the string and on its tension. The string is subjected to a vertical force of density f. The functions $u_0(x)$ and $v_0(x)$ denote respectively the initial displacement and the initial velocity of the string.

The change of variables

$$\omega_1 = \frac{\partial u}{\partial x}, \qquad \omega_2 = \frac{\partial u}{\partial t},$$

transforms (9.72) into the first-order system

$$\boxed{\frac{\partial \boldsymbol{\omega}}{\partial t} + \mathrm{A}\frac{\partial \boldsymbol{\omega}}{\partial x} = \mathbf{f}, \qquad x \in (a,b), \; t > 0} \tag{9.76}$$

where

$$\boldsymbol{\omega} = \begin{bmatrix} \omega_1 \\ \omega_2 \end{bmatrix}, \mathrm{A} = \begin{bmatrix} 0 & -1 \\ -c & 0 \end{bmatrix}, \mathbf{f} = \begin{bmatrix} 0 \\ f \end{bmatrix},$$

and the initial conditions are $\omega_1(x,0) = u_0'(x)$ and $\omega_2(x,0) = v_0(x)$ for $x \in (a,b)$.

In general, we can consider systems of the form (9.76) where $\boldsymbol{\omega}$, $\mathbf{f}$: $\mathbb{R} \times [0, \infty) \to \mathbb{R}^p$ are two given vector functions and $\mathrm{A} \in \mathbb{R}^{p \times p}$ is a matrix with constant coefficients. This system is said *hyperbolic* if A is diagonalizable and has real eigenvalues, that is, if there exists a nonsingular matrix $\mathrm{T} \in \mathbb{R}^{p \times p}$ such that

$$\mathrm{A} = \mathrm{T}\Lambda\mathrm{T}^{-1},$$

where $\Lambda = \mathrm{diag}(\lambda_1, ..., \lambda_p)$ is the diagonal matrix of the real eigenvalues of A, while $\mathrm{T} = (\mathbf{v}^1, \mathbf{v}^2, \dots, \mathbf{v}^p)$ is the matrix whose column vectors are the right eigenvectors of A. Thus

$$\mathrm{A}\mathbf{v}^k = \lambda_k \mathbf{v}^k, \qquad k = 1, \dots, p.$$

Introducing the *characteristic variables* $\mathbf{w} = \mathrm{T}^{-1}\boldsymbol{\omega}$, system (9.76) becomes

$$\frac{\partial \mathbf{w}}{\partial t} + \Lambda \frac{\partial \mathbf{w}}{\partial x} = \mathbf{g},$$

where $\mathbf{g} = \mathrm{T}^{-1}\mathbf{f}$. This is a system of p independent scalar equations of the form

$$\frac{\partial w_k}{\partial t} + \lambda_k \frac{\partial w_k}{\partial x} = g_k, \qquad k = 1, \dots, p.$$

When $g_k = 0$, its solution is given by $w_k(x,t) = w_k(x - \lambda_k t, 0)$, $k = 1, \dots, p$. Therefore the solution $\boldsymbol{\omega} = \mathrm{T}\mathbf{w}$ of problem (9.76) (for $\mathbf{f} = \mathbf{0}$) can be written as

$$\boldsymbol{\omega}(x,t) = \sum_{k=1}^{p} w_k(x - \lambda_k t, 0)\mathbf{v}^k.$$

The curve $(x_k(t), t)$ in the plane (x,t) that satisfies $x_k'(t) = \lambda_k$ is the kth characteristic curve (see Section 9.3) and w_k is constant along it. Then $\boldsymbol{\omega}(\overline{x}, \overline{t})$ depends only on the initial datum at the points $\overline{x} - \lambda_k \overline{t}$. For this reason, the set of p points that form the feet of the characteristics issuing from the point $(\overline{x}, \overline{t})$,

$$D(\overline{t}, \overline{x}) = \{x \in \mathbb{R} \ : \ x = \overline{x} - \lambda_k \overline{t} \ , \ k = 1, ..., p\}, \qquad (9.77)$$

is called the *domain of dependence* of the solution $\boldsymbol{\omega}(\overline{x}, \overline{t})$.

If (9.76) is set on a bounded interval (a, b) instead of on the whole real line, the inflow point for each characteristic variable w_k is determined by the sign of λ_k. Correspondingly, the number of positive eigenvalues determines the number of boundary conditions that should be assigned at $x = a$, whereas at $x = b$ the number of conditions that must be assigned equals the number of negative eigenvalues.

Example 9.6 System (9.76) is hyperbolic since A is diagonalizable with matrix

$$\mathrm{T} = \begin{bmatrix} -\dfrac{1}{\sqrt{c}} & \dfrac{1}{\sqrt{c}} \\ 1 & 1 \end{bmatrix}$$

and features two distinct real eigenvalues $\pm\sqrt{c}$ (representing the propagation velocities of the wave). Moreover, one boundary condition needs to be prescribed at every end-point, as in (9.75). ■

9.4.1 Finite difference approximation of the wave equation

To discretize in time equation (9.72) we can use the Newmark method formerly proposed in Chapter 8 for second-order ordinary differential equations, see (8.71). Still denoting by Δt the (uniform) time-step and using in space the classical finite difference method on a grid with nodes $x_j = x_0 + j\Delta x$, $j = 0, \ldots, N+1$, $x_0 = a$ and $x_{N+1} = b$, the Newmark scheme for (9.72) reads as follows: for any $n \geq 1$ find $\{u_j^n, v_j^n, j = 1, \ldots, N\}$ such that

$$\begin{aligned} u_j^{n+1} &= u_j^n + \Delta t v_j^n \\ &\quad + \Delta t^2 \left[\zeta(cw_j^{n+1} + f(x_j, t^{n+1})) + (1/2 - \zeta)(cw_j^n + f(x_j, t^n))\right], \qquad (9.78) \\ v_j^{n+1} &= v_j^n + \Delta t \left[(1-\theta)(cw_j^n + f(x_j, t^n)) + \theta(cw_j^{n+1} + f(x_j, t^{n+1}))\right], \end{aligned}$$

with $u_j^0 = u_0(x_j)$ and $v_j^0 = v_0(x_j)$ and $w_j^k = (u_{j+1}^k - 2u_j^k + u_{j-1}^k)/(\Delta x)^2$ for $k = n$ or $k = n+1$. System (9.78) must be completed by imposing the boundary conditions (9.75).

The Newmark method is implemented in Program 9.4. The input parameters are the vectors `xspan=[a,b]` and `tspan=[0,T]`, the number of discretization intervals in space (`nstep(1)`) and in time (`nstep(2)`), the scalar `c` (corresponding to the positive constant c), the function handles `u0` and `v0` associated with the initial data $u_0(x)$ and $v_0(x)$, respectively, and the function handles `g` and `fun` associated with the functions $g(x,t)$ and $f(x,t)$, respectively. Finally, the vector `param` allows to specify the values of the coefficients (`param(1)=`θ, `param(2)=`ζ). This method is second order accurate with respect to Δt if $\theta = 1/2$, whereas it is first order if $\theta \neq 1/2$. Moreover, the condition $\theta \geq 1/2$ is necessary to ensure stability (see Section 8.9).

Program 9.4. newmarkwave: Newmark method for the wave equation

```
function [xh,uh]=newmarkwave(xspan,tspan,nstep,param,...
                    c,u0,v0,g,f,varargin)
%NEWMARKWAVE solves the wave equation with the Newmark
```

```
% method.
% [XH,UH]=NEWMARKWAVE(XSPAN,TSPAN,NSTEP,PARAM,C,...
% U0,V0,G,F)
% solves the wave equation D^2 U/DT^2 - C D^2U/DX^2 = F
% in (XSPAN(1),XSPAN(2)) X (TSPAN(1),TSPAN(2)) using
% Newmark method with initial conditions U(X,0)=U0(X),
% DU/DX(X,0)=V0(X) and Dirichlet boundary conditions
% U(X,T)=G(X,T) for X=XSPAN(1) and X=XSPAN(2). C is a
% positive constant.
% NSTEP(1) is the number of space integration intervals
% NSTEP(2) is the number of time-integration intervals.
% PARAM(1)=ZETA and PARAM(2)=THETA.
% U0(X), V0(X), G(X,T) and F(x,T) are function handles.
% XH contains the nodes of the discretization.
% UH contains the numerical solutions at time TSPAN(2).
% [XH,UH]=NEWMARKWAVE(XSPAN,TSPAN,NSTEP,PARAM,C,...
% U0,V0,G,F,P1,P2,...) passes the additional parameters
%  P1,P2,...to the functions U0,V0,G,F.
h  = (xspan(2)-xspan(1))/nstep(1);
dt = (tspan(2)-tspan(1))/nstep(2);
zeta = param(1);  theta = param(2);
N = nstep(1)+1;
e = ones(N,1); D = spdiags([e -2*e e],[-1,0,1],N,N);
I = speye(N); lambda = dt/h;
A = I-c*lambda^2*zeta*D;
An = I+c*lambda^2*(0.5-zeta)*D;
A(1,:) = 0; A(1,1) = 1; A(N,:) = 0; A(N,N) = 1;
xh = (linspace(xspan(1),xspan(2),N))';
fn = f(xh,tspan(1),varargin{:});
un = u0(xh,varargin{:});
vn = v0(xh,varargin{:});
[L,U]=lu(A);
alpha = dt^2*zeta; beta = dt^2*(0.5-zeta);
theta1 = 1-theta;
for t = tspan(1)+dt:dt:tspan(2)
    fn1 = f(xh,t,varargin{:});
    rhs = An*un+dt*I*vn+alpha*fn1+beta*fn;
    temp = g([xspan(1),xspan(2)],t,varargin{:});
    rhs([1,N]) = temp;
    uh = L\rhs;    uh = U\uh;
    v = vn + dt*((1-theta)*(c*D*un/h^2+fn)+...
        theta*(c*D*uh/h^2+fn1));
    fn = fn1;    un = uh;    vn = v;
end
```

An alternative to the Newmark method is provided by the following *Leap-Frog* method

$$u_j^{n+1} - 2u_j^n + u_j^{n-1} = c\left(\frac{\Delta t}{\Delta x}\right)^2 (u_{j+1}^n - 2u_j^n + u_{j-1}^n), \tag{9.79}$$

which is obtained by discretizing both time and space derivatives by the centered finite difference formula (9.12).

Both Newmark (9.78) and *Leap-Frog* (9.79) schemes are second order accurate with respect to Δt and Δx. About stability, the *Leap-Frog* method is stable provided that the CFL condition $\Delta t \leq \Delta x/\sqrt{c}$ is

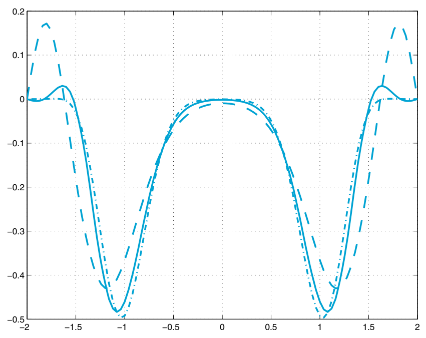


Figure 9.16. Comparison between the solutions obtained using the Newmark method for a discretization with $\varDelta x = 0.04$ and $\varDelta t = 0.15$ (*dashed line*), $\varDelta t = 0.075$ (*solid line*) and $\varDelta t = 0.0375$ (*dashed-dotted line*)

satisfied, while the Newmark method is unconditionally stable if $2\zeta \geq \theta \geq \frac{1}{2}$ (see [Joh90]).

Example 9.7 Using Program 9.4 we study the evolution of the initial condition $u_0(x) = e^{-10x^2}$ for $x \in (-2, 2)$, by putting $f = 0$ and $c = 1$ in (9.72). We assume $v_0 = 0$ and homogeneous Dirichlet boundary conditions. In Figure 9.16 we compare the solutions obtained at time $t = 3$ using $\varDelta x = 0.04$ and time-steps $\varDelta t = 0.15$ (*dashed line*), $\varDelta t = 0.075$ (*solid line*) and $\varDelta t = 0.0375$ (*dashed-dotted line*). The parameters of the Newmark method are $\theta = 1/2$ and $\zeta = 0.25$, and they ensure a second order unconditionally stable method. ■

Example 9.8 (Communications) In this example we use the equation (9.9) to model the way a telegraph wire transmits a pulse of voltage. The equation is a combination of diffusion and wave equations, and accounts for effects of finite velocity in a standard mass transport equation. In Figure 9.17 we compare the evolution of one bump (precisely a cubic B-spline (see [QSS07, Sect. 8.7.2])) centered in $x = 3$ and non-null in the interval (1,5) using the wave equation (9.72) (*dashed line*) and the telegrapher's equation (9.9) (*solid line*), on the interval $(0, 10)$ with $c = 1$, $\alpha = 0.5$ and $\beta = 0.04$. The initial speed is chosen to be $v_0(x) = -cu_0'(x)$ ($v_0(x) = -cu_0'(x) - \alpha/2u_0(x)$, resp.) for the wave (telegrapher's, resp.) equation, so that the bump travels with speed c. We have solved both the wave equation and telegrapher's equation by the Newmark scheme using $\varDelta x = 0.025$, time-step $\varDelta t = 0.1$, $\zeta = 1/4$ and $\theta = 1/2$. To approximate the wave equation we have called Program 9.4, while to solve the telegrapher's equation we have written a different program implementing the Newmark scheme (8.71) applied to equation (9.9). The presence of the dissipation effect is evident in the solution of the telegrapher's equation. ■

An alternative approach consists of discretizing the first-order system (9.76) instead of the (equivalent) second order scalar equation (9.72). When $\mathbf{f} = \mathbf{0}$, Lax-Wendroff and upwind schemes for the hyperbolic system (9.76) are defined as follows:

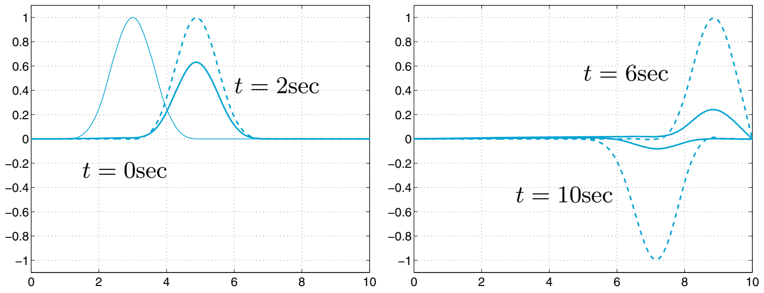


Figure 9.17. Propagation of a pulse of voltage using the wave equation (*dashed line*) and the telegrapher's equation (*solid line*). At left, the thin solid line represents the initial condition $u_0(x)$

1. *Lax-Wendroff method*

$$\begin{aligned}\boldsymbol{\omega}_j^{n+1} &= \boldsymbol{\omega}_j^n - \frac{\lambda}{2}\mathrm{A}(\boldsymbol{\omega}_{j+1}^n - \boldsymbol{\omega}_{j-1}^n)\\ &+\frac{\lambda^2}{2}\mathrm{A}^2(\boldsymbol{\omega}_{j+1}^n - 2\boldsymbol{\omega}_j^n + \boldsymbol{\omega}_{j-1}^n),\end{aligned} \tag{9.80}$$

2. *upwind (or forward Euler/decentered) method*

$$\begin{aligned}\boldsymbol{\omega}_j^{n+1} &= \boldsymbol{\omega}_j^n - \frac{\lambda}{2}\mathrm{A}(\boldsymbol{\omega}_{j+1}^n - \boldsymbol{\omega}_{j-1}^n)\\ &+\frac{\lambda}{2}|\mathrm{A}|(\boldsymbol{\omega}_{j+1}^n - 2\boldsymbol{\omega}_j^n + \boldsymbol{\omega}_{j-1}^n),\end{aligned} \tag{9.81}$$

   where $|\mathrm{A}| = \mathrm{T}|\Lambda|\mathrm{T}^{-1}$ and $|\Lambda|$ is the diagonal matrix of the moduli of the eigenvalues of A.

The Lax-Wendroff method is second order accurate (in both time and space), while the upwind scheme is first order.

About stability, all considerations made in Section 9.3.1 are still valid, provided the CFL condition (9.65) is replaced by

$$\Delta t < \frac{\Delta x}{\rho(\mathrm{A})}. \tag{9.82}$$

As usual, $\rho(\mathrm{A})$ denotes the spectral radius of A.

For the proof of these results see, e.g., [QV94], [LeV02], [GR96], [QSS07, Chapter 13].

See Exercises 9.8-9.9.

Let us summarize

1. One-dimensional boundary value problems are set up on an interval; boundary conditions on the solution (or on its derivative) must be prescribed at the endpoints of the interval;
2. numerical approximation can be carried out by finite-differences (arising from truncated Taylor series) or by finite-elements (arising from the weak formulation of the differential problem; in this context, both test and trial functions are piecewise polynomials);
3. multidimensional problems can be faced by using similar arguments. For two-dimensional boundary-value problems, finite element approximations make use of piecewise polynomials, where "piecewise" refers to either triangles or quadrilaterals of a grid partitioning the spatial domain;
4. matrices arising from both finite element and finite difference discretizations are sparse and ill-conditioned;
5. initial-boundary-value problems contain time derivatives of the solution which are discretized by finite difference formulas, of either explicit or implicit type;
6. when explicit schemes are used, stability conditions have to be satisfied: the time-step turns out to be bounded by the spatial grid size. On the other hand, when implicit schemes are used, a linear algebraic system (similar to that obtained for stationary problems) has to be solved at each time level;
7. in this Chapter we have presented some simple linear problems of elliptic, parabolic and hyperbolic type. For a more exhaustive treatment of this subject we suggest the reader to refer to the bibliography presented in the next Section.

9.5 What we haven't told you

We could simply say that we have told you almost nothing, since the field of numerical analysis which is devoted to the numerical approximation of partial differential equations is so broad and multifaceted to deserve an entire monograph simply for addressing the most essential concepts (see, e.g., [TW98], [EEHJ96]).

We would like to mention that the finite element method is nowadays probably the most widely diffused method for the numerical solution of partial differential equations (see, e.g., [Qua13], [QV94], [Bra97], [BS01]). As already mentioned the MATLAB toolbox `pde` allows the solution of a broad family of partial differential equations by the linear finite element method, in particular for the discretization of space variables.

Other popular techniques are the spectral methods (see, e.g., [CHQZ06], [CHQZ07], [Fun92], [BM92], [KS99]) and the finite volume method (see, e.g., [Krö98], [Hir88] and [LeV02]).

Octave 9.1 The Octave-Forge package `bim` offers most of the main functionalities of the `pde` toolbox, although its syntax is in general not compatible with that of MATLAB. ■

9.6 Exercises

Exercise 9.1 Verify that matrix (9.15) is positive definite.

Exercise 9.2 Verify that the eigenvalues of the matrix A$\in \mathbb{R}^{N\times N}$, defined in (9.15), are

$$\lambda_j = 2(1-\cos(j\theta)), \quad j=1,\dots,N,$$

while the corresponding eigenvectors are

$$\mathbf{q}_j = (\sin(j\theta), \sin(2j\theta), \dots, \sin(Nj\theta))^T,$$

where $\theta = \pi/(N+1)$. Deduce that $K(A)$ is proportional to h^{-2}.

Exercise 9.3 Prove that the quantity (9.12) provides a second order approximation of $u''(\bar{x})$ with respect to h.

Exercise 9.4 Compute the matrix and the right-hand side of the numerical scheme that we have proposed to approximate problem (9.17).

Exercise 9.5 Use the finite difference method to approximate the boundary-value problem

$$\begin{cases} -u'' + \dfrac{k}{T}u = \dfrac{w}{T} \text{ in } (0,1), \\ u(0) = u(1) = 0, \end{cases}$$

where $u = u(x)$ represents the vertical displacement of a string of length 1, subject to a transverse load of intensity $w(x)$ per unit length. T is the tension and k is the elastic coefficient of the string. For the case in which $w(x) = 1 + \sin(4\pi x)$, $T = 1$ and $k = 0.1$, compute the solution corresponding to $h = 1/i$, with $i = 10, 20, 40$, and deduce the order of accuracy of the method.

Exercise 9.6 Use the finite difference method to solve problem (9.17) in the case where the following boundary conditions are prescribed at the endpoints (called *Neumann* boundary conditions)

$$u'(a) = \alpha, \qquad u'(b) = \beta.$$

Use the formulae given in (4.11) to discretize $u'(a)$ and $u'(b)$.

Exercise 9.7 Verify that, when using a uniform grid, the right-hand side of the system (9.14) associated with the centered finite difference scheme coincides, up a factor h, with that of the finite element scheme (9.27) provided that the composite trapezoidal formula is used to compute the integrals on the elements I_{j-1} and I_j.

Exercise 9.8 Verify that $\text{div}\nabla\phi = \Delta\phi$, where ∇ is the gradient operator that associates to a function u the vector whose components are the first order partial derivatives of u.

Exercise 9.9 (Thermodynamics) Consider a square plate whose side length is 20 cm and whose thermal conductivity is $k = 0.2$ cal/(sec·cm·C). Denote by $Q = 5$ cal/(cm^3·sec) the heat production rate per unit area. The temperature $T = T(x, y)$ of the plate satisfies the equation $-\Delta T = Q/k$. Assuming that T is null on three sides of the plate and is equal to 1 on the fourth side, determine the temperature T at the center of the plate.

Exercise 9.10 Verify that the solution of problem (9.72), (9.74) – (9.75) (with $f = 0$) satisfies the identity

$$\int_a^b (u_t(x,t))^2 \mathrm{d}x + c\int_a^b (u_x(x,t))^2 \mathrm{d}x = \tag{9.83}$$

$$\int_a^b (v_0(x))^2 \mathrm{d}x + c\int_a^b (u_{0,x}(x))^2 \mathrm{d}x,$$

provided that $u_0(a) = u_0(b) = 0$.

Exercise 9.11 Prove that the numerical solution provided by the backward Euler/centered scheme (9.62) is unconditionally stable, that is $\forall\Delta t > 0$,

$$\|\mathbf{u}^n\|_{\Delta,2} \leq \|\mathbf{u}^0\|_{\Delta,2} \qquad \forall n \geq 0. \tag{9.84}$$

Exercise 9.12 Prove that the solution provided by the upwind scheme (9.59) satisfies the estimate

$$\|\mathbf{u}^n\|_{\Delta,\infty} \leq \|\mathbf{u}^0\|_{\Delta,\infty} \qquad \forall n \geq 0, \tag{9.85}$$

provided that the CFL condition has been verified. The inequality (9.85) is named *discrete maximum principle.*

Exercise 9.13 Solve problem (9.47) with $a = 1$, $x \in (0, 0.5)$, $t \in (0, 1)$, initial datum $u^0(x) = 2\cos(4\pi x) + \sin(20\pi x)$ and boundary condition $u(0,t) = 2\cos(4\pi t) - sin(20\pi t)$ for $t \in (0, 1)$. Use both Lax-Wendroff (9.57) and upwind (9.59) schemes. Set the CFL number equal to 0.5. Verify experimentally that the Lax-Wendroff scheme is second-order accurate with respect to Δx and Δt, while the upwind scheme is first-order accurate. To evaluate the error use the norm $\|\cdot\|_{\Delta,2}$.

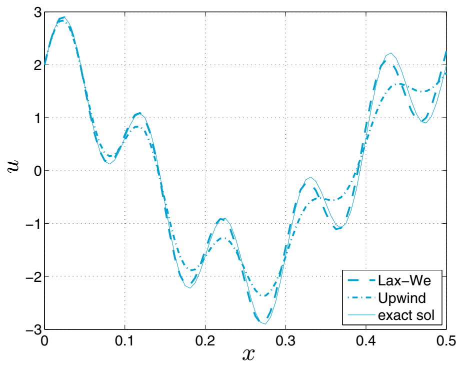


Figure 9.18. Numerical solutions at time $t = 5$ for the problem (9.47) by using data of Exercise 9.13. The CFL number is 0.8

Exercise 9.14 In Figure 9.18 both exact and numerical solutions of problem (9.47) at time $t = 5$ are shown. The latter are computed by the Lax-Wendroff (9.57) and upwind (9.59) schemes, using the same data of Exercise 9.13. By knowing that the CFL number is 0.8 and that we have used $\Delta t = 5.e - 3$, comment on the dissipation and dispersion coefficients that we have obtained.

10 Solutions of the exercises

In this chapter we will provide solutions of the exercises that we have proposed at the end of the previous eight chapters. The expression "Solution n.m" is an abridged notation for "Solution of Exercise n.m" (mth Exercise of the nth Chapter).

10.1 Chapter 1

Exercise 1.1 Only the numbers of the form $\pm 0.1a_2 \cdot 2^e$ with $a_2 = 0, 1$ and $e = \pm 2, \pm 1, 0$ belong to the set $\mathbb{F}(2, 2, -2, 2)$. For a given exponent, we can represent in this set only the two numbers 0.10 and 0.11, and their opposites. Consequently, the number of elements belonging to $\mathbb{F}(2, 2, -2, 2)$ is 20. Finally, $\epsilon_M = 1/2$.

Exercise 1.2 For any fixed exponent, each of the digits $a_2, \ldots, a_t$ can assume β different values, while a_1 can assume only $\beta - 1$ values. Therefore $2(\beta - 1)\beta^{t-1}$ different numbers can be represented (the 2 accounts for the positive and negative sign). On the other hand, the exponent can assume $U - L + 1$ values. Thus, the set $\mathbb{F}(\beta, t, L, U)$ contains $2(\beta - 1)\beta^{t-1}(U - L + 1)$ different elements.

Exercise 1.3 Thanks to the Euler formula $i = e^{i\pi/2}$; we obtain $i^i = e^{-\pi/2}$, that is, a real number. In MATLAB

```
exp(-pi/2)
ans =
    0.2079
i^i
ans =
    0.2079
```

Exercise 1.4 Use the instructions `U=2*eye(10)-3*diag(ones(8,1),2)` and `L=2*eye(10)-3*diag(ones(8,1),-2)`.

Exercise 1.5 We can interchange the third and seventh rows of the previous matrix using the instructions: `r=[1:10]; r(3)=7; r(7)=3; Lr=L(r,:)`. Notice that the character `:` in `L(r,:)` ensures that all columns of `L` are spanned in the usual increasing order (from the first to the last). To interchange the fourth column with the eighth column we can write `c=[1:10]; c(8)=4; c(4)=8; Lc=L(:,c)`. Similar instructions can be used for the upper triangular matrix.

`L(r,:)`

Exercise 1.6 We can define the matrix `A = [v1;v2;v3;v4]` where `v1`, `v2`, `v3` and `v4` are the 4 given row vectors. They are linearly independent iff the determinant of `A` is different from 0, which is not true in our case.

Exercise 1.7 The two given functions f and g have the symbolic expression:

```
syms x
f=sqrt(x^2+1); pretty(f)
```

$$(x^2+1)^{1/2}$$

```
g=sin(x^3)+cosh(x); pretty(g)
```

$$\mathtt{sin}(x^3)+\mathtt{cosh}(x)$$

`pretty`

The command `pretty(f)` prints the symbolic expression `f` in a format that resembles type-set mathematics. At this stage, the symbolic expression of the first and second derivatives and the integral of f can be obtained with the following instructions:

```
diff(f,x)
ans =
1/(x^2+1)^(1/2)*x
diff(f,x,2)
ans =
-1/(x^2+1)^(3/2)*x^2+1/(x^2+1)^(1/2)
int(f,x)
ans =
1/2*x*(x^2+1)^(1/2)+1/2*asinh(x)
```

Similar instructions can be used for the function g.

Exercise 1.8 The accuracy of the computed roots downgrades as the polynomial degree increases. This experiment reveals that the accurate computation of the roots of a polynomial of high degree can be troublesome.

Exercise 1.9 Here is a possible program to compute the sequence:

```
function I=sequence(n)
I = zeros(n+2,1); I(1) = (exp(1)-1)/exp(1);
for i = 0:n, I(i+2) = 1 - (i+1)*I(i+1); end
```

The sequence computed by this program doesn't tend to zero (as `n` increases), but it diverges with alternating sign. This behavior is a direct consequence of rounding errors propagation.

Exercise 1.10 The anomalous behavior of the computed sequence is due to the propagation of roundoff errors from the innermost operation. In particular, when $4^{1-n}z_n^2$ is less than $\epsilon_M/2$, the subsequent element z_{n+1} of the sequence is equal to 0. This happens for $n \geq 30$.

Exercise 1.11 The proposed method is a special instance of the Monte Carlo method and is implemented by the following program:

```
function mypi=pimontecarlo(n)
x = rand(n,1); y = rand(n,1);
z = x.^2+y.^2;
v = (z <= 1);
m=sum(v); mypi=4*m/n;
```

The command `rand` generates a sequence of pseudo-random numbers. The instruction `v = (z <= 1)` is a shortand version of the following procedure: we check whether `z(k) <= 1` for any component of the vector `z`. If the inequality is satisfied for the `k`th component of `z` (that is, the point `(x(k),y(k))` belongs to the interior of the unit circle) `v(k)` is set equal to 1, and to 0 otherwise. The command `sum(v)` computes the sum of all components of `v`, that is, the number of points falling in the interior of the unit circle. `sum`

By launching the program `mypi=pimontecarlo(n)` for different values of `n`, when `n` increases, the approximation `mypi` of π becomes more accurate. For instance, for `n=1000` we obtain `mypi=3.1120`, whilst for `n=300000` we have `mypi=3.1406`. (Obviously, since the numbers are randomly generated, the result obtained with the same value of `n` may change at each run.)

Exercise 1.12 To answer the question we can use the following *function*:

```
function pig=bbpalgorithm(n)
pig = 0;
for m=0:n
  m8 = 8*m;
  pig = pig + (1/16)^m*(4/(m8+1)-(2/(m8+4)+ ...
        1/(m8+5)+1/(m8+6)));
end
```

For `n=10` we obtain an approximation `pig` of π that coincides (up to MATLAB precision) with the persistent MATLAB variable `pi`. In fact, this algorithm is extremely efficient and allows the rapid computation of hundreds of significant digits of π.

Exercise 1.13 The binomial coefficient can be computed by the following program (see also the MATLAB function `nchoosek`): `nchoosek`

```
function bc=bincoeff(n,k)
k = fix(k); n = fix(n);
if k > n, disp('k must be between  0 and n');
   return; end
if k > n/2, k = n-k; end
if k <= 1,  bc = n^k; else
  num = (n-k+1):n; den = 1:k; el = num./den;
  bc = prod(el);
end
```

The command `fix(k)` rounds `k` to the nearest integer smaller than `k`. The command `disp(string)` displays the string, without printing its name. The command `return` terminates the execution of the function. Finally, `prod(el)` computes the product of all elements of the vector `el`. `fix` `return` `prod`

Exercise 1.14 The following *functions* compute f_n using the form $f_i = f_{i-1} + f_{i-2}$ (`fibrec`) or using the form (1.14) (`fibmat`):

```
function f=fibrec(n)
if n == 0
    f = 0;
elseif n == 1
    f = 1;
else
    f = fibrec(n-1)+fibrec(n-2);
end

function f=fibmat(n)
f = [0;1];
A = [1 1; 1 0];
f = A^n*f;
f = f(1);
```

For `n=20` we obtain the following results:

```
t=cputime; fn=fibrec(20), cpu=cputime-t
fn =
        6765
cpu =
    0.48
t=cputime; fn=fibmat(20), cpu=cputime-t
fn =
        6765
cpu =
     0
```

The recursive *function* `fibrec` requires much more CPU time than `fibmat`. The latter requires to compute only the power of a matrix, an easy operation in MATLAB.

10.2 Chapter 2

Exercise 2.1 The command `fplot` allows us to study the graph of the given function f for various values of γ. For $\gamma = 1$, the corresponding function does not have real zeros. For $\gamma = 2$, there is only one zero, $\alpha = 0$, with multiplicity equal to four (that is, $f(\alpha) = f'(\alpha) = f''(\alpha) = f'''(\alpha) = 0$, while $f^{(4)}(\alpha) \neq 0$). Finally, for $\gamma = 3$, f has two distinct zeros, one in the interval $(-3, -1)$ and the other one in $(1, 3)$. In the case $\gamma = 2$, the bisection method cannot be used since it is impossible to find an interval (a, b) in which $f(a)f(b) < 0$. For $\gamma = 3$, starting from the interval $[a, b] = [-3, -1]$, the bisection method (Program 2.1) converges in 34 iterations to the value $\alpha = -1.85792082914850$ (with $f(\alpha) \simeq -3.6 \cdot 10^{-12}$), using the following instructions:

```
f=@(x) cosh(x)+cos(x)-3; a=-3; b=-1;
tol=1.e-10; nmax=200;
[zero,res,niter]=bisection(f,a,b,tol,nmax)

zero =
   -1.8579
res =
    -3.6872e-12
niter =
    34
```

Similarly, choosing `a=1` and `b=3`, for $\gamma = 3$ the bisection method converges after 34 iterations to the value $\alpha = 1.8579208291485$ with $f(\alpha) \simeq -3.6877 \cdot 10^{-12}$.

Exercise 2.2 We have to compute the zeros of the function $f(V) = pV + aN^2/V - abN^3/V^2 - pNb - kNT$, where N is the number of molecules. Plotting the graph of f, we see that this function has just a simple zero in the interval $(0.01, 0.06)$ with $f(0.01) < 0$ and $f(0.06) > 0$. We can compute this zero using the bisection method as follows:

```
f=@(x) 35000000*x+401000./x-17122.7./x.^2-1494500;
[zero,res,niter]=bisection(f,0.01,0.06,1.e-12,100)

zero =
    0.0427
res =
   -6.3814e-05
niter =
    35
```

Exercise 2.3 The unknown value of ω is the zero of the function $f(\omega) = s(1, \omega) - 1 = 9.8[\sinh(\omega) - \sin(\omega)]/(2\omega^2) - 1$. From the graph of f we conclude that f has a unique real zero in the interval $(0.5, 1)$. Starting from this interval, the bisection method computes the value $\omega = 0.61214447021484$ with the desired tolerance in 15 iterations as follows:

```
f=@(omega) 9.8/2*(sinh(omega)-sin(omega))./omega.^2-1;
[zero,res,niter]=bisection(f,0.5,1,1.e-05,100)
zero =
   6.1214e-01
res =
   3.1051e-06
niter =
    15
```

Exercise 2.4 The inequality (2.6) can be derived by observing that $|e^{(k)}| < |I^{(k)}|/2$ with $|I^{(k)}| < \frac{1}{2}|I^{(k-1)}| < 2^{-k-1}(b-a)$. Consequently, the error at the iteration k_{min} is less than ε if k_{min} is such that $2^{-k_{min}-1}(b-a) < \varepsilon$, that is, $2^{-k_{min}-1} < \varepsilon/(b-a)$, which proves (2.6).

Exercise 2.5 The implemented formula is less sensitive to roundoff errors.

Exercise 2.6 In Solution 2.1 we have analyzed the zeros of the given function with respect to different values of γ. Let us consider the case when $\gamma = 2$. Starting from the initial guess $x^{(0)} = 1$, the Newton method (Program 2.2) converges to the value $\bar{\alpha} = 1.4961e-4$ in 31 iterations with `tol=1.e-10` while the exact zero of f is equal to 0. This discrepancy is due to the fact that f is almost a constant in a neighborhood of its zero, whence the root-finding is an ill-conditioned problem (see the comment at the end of Sect. 2.8.2). The method converges at the same solution and with the same number of iterations even if we set `tol=`ϵ_M. Actually, the corresponding residual computed by MATLAB is 0. Let us set now $\gamma = 3$. The Newton method with `tol=`$\epsilon_M$converges to the value 1.85792082915020 in 9 iterations starting from $x^{(0)} = 1$, while if $x^{(0)} = -1$ after 9 iterations it converges to the value -1.85792082915020 (in both cases the residuals are zero in MATLAB).

Exercise 2.7 The square and the cube roots of a number a are the solutions of the equations $x^2 = a$ and $x^3 = a$, respectively. Thus, the corresponding algorithms are: for a given $x^{(0)}$ compute

$$x^{(k+1)} = \frac{1}{2}\left(x^{(k)} + \frac{a}{x^{(k)}}\right),\ k \geq 0 \qquad \text{for the square root,}$$

$$x^{(k+1)} = \frac{1}{3}\left(2x^{(k)} + \frac{a}{(x^{(k)})^2}\right),\ k \geq 0 \text{ for the cube root.}$$

Exercise 2.8 Setting $\delta x^{(k)} = x^{(k)} - \alpha$, from the Taylor expansion of f we find:

$$0 = f(\alpha) = f(x^{(k)}) - \delta x^{(k)} f'(x^{(k)}) + \frac{1}{2}(\delta x^{(k)})^2 f''(x^{(k)}) + \mathcal{O}((\delta x^{(k)})^3). \quad (10.1)$$

The Newton method yields

$$\delta x^{(k+1)} = \delta x^{(k)} - f(x^{(k)})/f'(x^{(k)}). \quad (10.2)$$

Combining (10.1) with (10.2), we have

$$\delta x^{(k+1)} = \frac{1}{2}(\delta x^{(k)})^2 \frac{f''(x^{(k)})}{f'(x^{(k)})} + \mathcal{O}((\delta x^{(k)})^3).$$

After division by $(\delta x^{(k)})^2$ and letting $k \to \infty$ we prove the convergence result.

Exercise 2.9 For certain values of β the equation (2.2) can have two roots that correspond to different configurations of the rods system. The two initial values that are suggested have been chosen conveniently to allow the Newton method to converge toward one or the other root, respectively. We solve the problem for $\beta = k\pi/150$ with $k = 0, \dots, 100$ (if $\beta > 2.6389$ the Newton method does not converge since the system has no admissible configuration). We use the following instructions to obtain the solution of the problem (shown in Figure 10.1, left):

```
a1=10; a2=13; a3=8; a4=10;
ss = (a1^2 + a2^2 - a3^2+ a4^2)/(2*a2*a4);
n=150; x01=-0.1; x02=2*pi/3; nmax=100;
beta=zeros(100,1);
for k=0:100
   w = k*pi/n; i=k+1; beta(i) = w;
   f  = @(x) 10/13*cos(w)-cos(x)-cos(w-x)+ss;
   df = @(x) sin(x)-sin(w-x);
   [zero,res,niter]=newton(f,df,x01,1e-5,nmax);
   alpha1(i) = zero; niter1(i) = niter;
   [zero,res,niter]=newton(f,df,x02,1e-5,nmax);
   alpha2(i) = zero; niter2(i) = niter;
end
plot(beta,alpha1,'c--',beta,alpha2,'c','Linewidth',2)
grid on
```

The components of the vectors `alpha1` and `alpha2` are the angles computed for different values of β, while the components of the vectors `niter1` and `niter2` are the number of Newton iterations (between 2 and 6) necessary to compute the zeros with the requested tolerance.

Exercise 2.10 From an inspection of its graph we see that f has two positive real zeros ($\alpha_2 \simeq 1.5$ and $\alpha_3 \simeq 2.5$) and one negative ($\alpha_1 \simeq -0.5$). The Newton method converges in 4 iterations (having set $x^{(0)} = -0.5$ and `tol = 1.e-10`) to the value α_1:

```
f=@(x) exp(x)-2*x^2; df=@(x) exp(x)-4*x;
x0=-0.5; tol=1.e-10; nmax=100;
format long; [zero,res,niter]=newton(f,df,x0,tol,nmax)

zero =
  -0.53983527690282
res =
     0
niter =
     4
```

The given function has a maximum at $\bar{x} \simeq 0.3574$ (which can be obtained by applying the Newton method to the function f'): for $x^{(0)} < \bar{x}$ the method converges to the negative zero. If $x^{(0)} = \bar{x}$ the Newton method cannot be applied since $f'(\bar{x}) = 0$. For $x^{(0)} > \bar{x}$ the method converges to one of the two positive zeros, either α_2 or α_3.

Exercise 2.11 Let us set $x^{(0)} = 0$ and `tol=` ϵ_M. In MATLAB the Newton method converges in 43 iterations to the value 0.641182985886554, while in Octave it converges in 32 iterations to the value 0.641184396264531. By taking the MATLAB approximated value as the reference solution in our error analysis, we can observe that the (approximate) errors decrease only linearly when k increases (see Figure 10.1, right). This behavior is due to the fact that α has a multiplicity greater than 1. To recover a second-order method we can use the modified Newton method.

Exercise 2.12 We should compute the zero of the function $f(x) = \sin(x) - \sqrt{2gh/v_0^2}$. By inspecting its graph, we can conclude that f has one zero in the interval $(0, \pi/2)$. The Newton method with $x^{(0)} = \pi/4$ and `tol=` 10^{-10} converges in 5 iterations to the value 0.45862863227859.

Exercise 2.13 Using the data given in the exercise, the solution can be obtained with the following instructions:

```
M=6000; v=1000; f=@(r) M-v*(1+r)./r.*((1+r).^5-1);
df=@(r) v*((1+r).^5.*(1-5*r)-1)./(r.^2);
[zero,res,niter]=bisection(f,0.01,0.1,1.e-12,5);
[zero,res,niter]=newton(f,df,zero,1.e-12,100)
```

The Newton method converges to the desired result in 3 iterations.

Exercise 2.14 By a graphical study, we see that (2.38) is satisfied for a value of α in $(\pi/6, \pi/4)$. Using the following instructions:

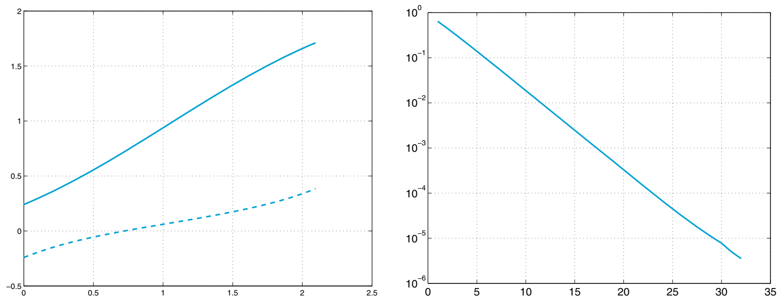


Figure 10.1. At left: the two curves represent the two possible configurations of roads system in terms of the angle α versus $\beta \in [0, 2\pi/3]$ (Solution 2.9). At right: error versus iteration number of the Newton method for the computation of the zero of the function $f(x) = x^3 - 3x^2 2^{-x} + 3x4^{-x} - 8^{-x}$ (Solution 2.11)

```
l1=8; l2=10; g=3*pi/5;
f=@(a) -l2*cos(g+a)/sin(g+a)^2-l1*cos(a)/sin(a)^2;
df=@(a) [l2/sin(g+a)+2*l2*cos(g+a)^2/sin(g+a)^3+...
         l1/sin(a)+2*l1*cos(a)^2/sin(a)^3];
[zero,res,niter]=newton(f,df,pi/4,1.e-15,100)
L=l2/sin(2*pi/5-zero)+l1/sin(zero)
```

the Newton method provides the approximate value 0.59627992746547 in 6 iterations, starting from $x^{(0)} = \pi/4$. We deduce that the maximum length of a rod that can pass in the corridor is $L = 30.5484$.

Exercise 2.15 If α is a zero of f with multiplicity m, then there exists a function h such that $h(\alpha) \neq 0$ and $f(x) = h(x)(x - \alpha)^m$. By computing the first derivative of the iteration function ϕ_N of the Newton method, we have

$$\phi_N'(x) = 1 - \frac{[f'(x)]^2 - f(x)f''(x)}{[f'(x)]^2} = \frac{f(x)f''(x)}{[f'(x)]^2}.$$

By replacing f, f' and f'' with the corresponding expressions as functions of $h(x)$ and $(x - \alpha)^m$, we obtain $\lim_{x\to\alpha} \phi_N'(x) = 1 - 1/m$, hence $\phi_N'(\alpha) = 0$ if and only if $m = 1$. Consequently, if $m = 1$ the method converges at least quadratically, according to (2.9). If $m > 1$ the method converges with order 1 according to Proposition 2.1.

Exercise 2.16 Let us inspect the graph of f by using the following commands:

```
f=@(x) x^3+4*x^2-10; fplot(f,[-10,10]); grid on;
fplot(f,[-5,5]); grid on;
fplot(f,[0,2]); grid on; axis([0,2,-10,15])
```

we can see that f has only one real zero, equal approximately to 1.36 (see Figure 10.2, left, for the last graph generated by the previous instructions). The iteration function and its derivative are:

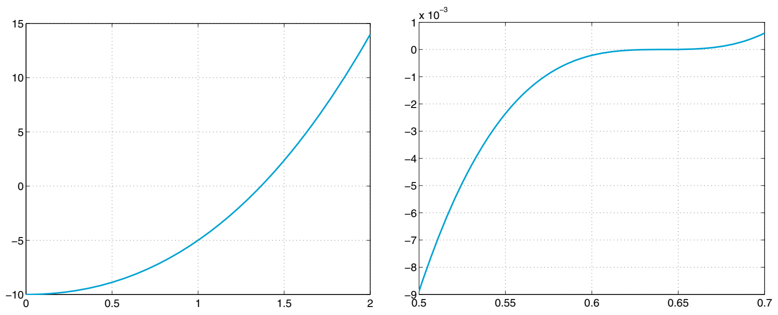


Figure 10.2. At left: graph of $f(x) = x^3 + 4x^2 - 10$ for $x \in [0, 2]$ (Solution 2.16). At right: graph of $f(x) = x^3 - 3x^2 2^{-x} + 3x4^{-x} - 8^{-x}$ for $x \in [0.5, 0.7]$ (Solution 2.18)

$$\begin{aligned}
\phi(x) &= \frac{2x^3 + 4x^2 + 10}{3x^2 + 8x} = -\frac{f(x)}{3x^2 + 8x} + x,\\
\phi'(x) &= \frac{(6x^2 + 8x)(3x^2 + 8x) - (6x + 8)(2x^3 + 4x^2 + 10)}{(3x^2 + 8x)^2}\\
&= \frac{(6x + 8)f(x)}{(3x^2 + 8x)^2},
\end{aligned}$$

and $\phi(\alpha) = \alpha$. We easily deduce that $\phi'(\alpha) = 0$, since $f(\alpha) = 0$. Consequently, the proposed method converges (at least) quadratically.

Exercise 2.17 The proposed method is convergent at least with order 2 since $\phi'(\alpha) = 0$.

Exercise 2.18 By keeping the remaining parameters unchanged, the method converges after 30 iterations to the value 0.641182210863894 which differs by less than 10^{-7} from the result previously computed in Solution 2.11. However, the behavior of the function, which is quite flat near $x = 0$, suggests that the result computed previously could be more accurate. In Figure 10.2, right, we show the graph of f in $(0.5, 0.7)$, obtained by the following instructions:

```
f=@(x) x^3-3*x^2*2^(-x)+3*x*4^(-x)-8^(-x);
fplot(f,[0.5 0.7]);
grid on
```

10.3 Chapter 3

Exercise 3.1 Since $x \in (x_0, x_n)$, there exists an interval $I_i = (x_{i-1}, x_i)$ such that $x \in I_i$. We can easily see that $\max_{x \in I_i} |(x - x_{i-1})(x - x_i)| = h^2/4$. If we bound $|x - x_{i+1}|$ above by $2h$, $|x - x_{i-2}|$ by $3h$ and so on, we obtain the inequality (3.6).

Exercise 3.2 In all cases we have $n = 4$ and thus we should estimate the fifth derivative of each function in the given interval. We find: $\max_{x\in[-1,1]} |f_1^{(5)}| \simeq 1.18$, $\max_{x\in[-1,1]} |f_2^{(5)}| \simeq 1.54$, $\max_{x\in[-\pi/2,\pi/2]} |f_3^{(5)}| \simeq 1.41$. Thanks to formula (3.7), the upper bounds for the corresponding errors are about 0.0018, 0.0024 and 0.0211, respectively.

Exercise 3.3 Using the MATLAB command `polyfit` we compute the interpolating polynomials of degree 3 in the two cases:

```
year=[1975 1980 1985 1990];
west=[72.8 74.2 75.2 76.4];
east=[70.2 70.2 70.3 71.2];
cwest=polyfit(year,west,3);
ceast=polyfit(year,east,3);
estwest=polyval(cwest,[1977 1983 1988]);
esteast=polyval(ceast,[1977 1983 1988]);
```

The estimated values in 1977, 1983 and 1988 are

```
estwest =
   73.4464   74.8096   75.8576
esteast =
   70.2328   70.2032   70.6992
```

for the Western and Eastern Europe, respectively.

Exercise 3.4 We choose the month as time-unit. The initial time $t_0 = 1$ corresponds to November 1987, while $t_7 = 157$ to November 2000. With the following instructions we compute the coefficients of the polynomial interpolating the given prices:

```
time = [1 14 37 63 87 99 109 157];
price = [4.5 5 6 6.5 7 7.5 8 8];
[c] = polyfit(time,price,7);
```

Setting `[price2002]=polyval(c,181)` we find that the estimated price of the magazine in November 2002 is approximately 11.24 euros.

Exercise 3.5 In this special case, since the number of interpolation nodes is 4, the interpolatory cubic spline, computed by the command `spline`, coincides with the interpolating polynomial. As a matter of fact, the spline interpolates the nodal data, moreover its first and second derivatives are continuous while the third derivative is continuous at the internal nodes x_1 and x_2, thanks to the *not-a-knot* condition used by MATLAB. This wouldn't be true for the natural interpolating cubic spline.

Exercise 3.6 We use the following instructions:

```
T = [4:4:20];
rho=[1000.7794,1000.6427,1000.2805,999.7165,998.9700];
Tnew = [6:4:18]; format long e;
rhonew = spline(T,rho,Tnew)

rhonew =
  Columns 1 through 2
     1.000740787500000e+03     1.000488237500000e+03
  Columns 3 through 4
     1.000022450000000e+03     9.993649250000000e+02
```

The comparison with the further measures shows that the approximation is extremely accurate. Note that the state equation for the sea-water (UNESCO, 1980) assumes a fourth-order dependence of the density on the temperature. However, the coefficient of the fourth power of T is of the order of 10^{-9} and the cubic spline provides a good approximation of the measured values.

Exercise 3.7 We compare the results computed using the interpolatory cubic spline obtained using the MATLAB command `spline` (denoted with `s3`), the interpolatory natural spline (`s3n`) and the interpolatory spline with null first derivatives at the endpoints of the interpolatory interval (`s3d`) (computed with Program 3.2). We use the following instructions:

```
year=[1965 1970 1980 1985 1990 1991];
production=[17769 24001 25961 34336 29036 33417];
z=[1962:0.1:1992];
s3  = spline(year,production,z);
s3n = cubicspline(year,production,z);
s3d = cubicspline(year,production,z,0,[0 0]);
```

In the following table we resume the computed values (expressed in thousands of tons of goods):

year	1962	1977	1992
`s3`	514.6	2264.2	4189.4
`s3n`	1328.5	2293.4	3779.8
`s3d`	2431.3	2312.6	2216.6

The comparison with the real data (1238, 2740.3 and 3205.9 thousands of tons, respectively) shows that the values predicted by the natural spline are accurate also outside the interpolation interval (see Figure 10.3, left). On the contrary, the interpolating polynomial introduces large oscillations near this end-point and underestimates the production of as many as -7768.5×10^6 Kg for 1962.

Exercise 3.8 The interpolating polynomial `p` and the spline `s3` can be evaluated by the following instructions:

```
pert = 1.e-04;
x=[-1:2/20:1]; y=sin(2*pi*x)+(-1).^[1:21]*pert;
z=[-1:0.01:1]; c=polyfit(x,y,20);
p=polyval(c,z); s3=spline(x,y,z);
```

When we use the unperturbed data (`pert=0`) the graphs of both `p` and `s3` are indistinguishable from that of the given function. The situation changes dramatically when the perturbed data are used (`pert=1.e-04`). In particular, the interpolating polynomial shows strong oscillations at the end-points of the interval, whereas the spline remains practically unchanged (see Figure 10.3, right). This example shows that approximation by splines is in general more stable with respect to perturbation errors than the global Lagrange interpolation.

Exercise 3.9 If $n = m$, setting $\tilde{f} = \Pi_n f$ we find that the first member of (3.28) is null. Thus in this case $\Pi_n f$ is the solution of the least-squares problem. Since the interpolating polynomial is unique, we deduce that this is the unique solution to the least-squares problem.

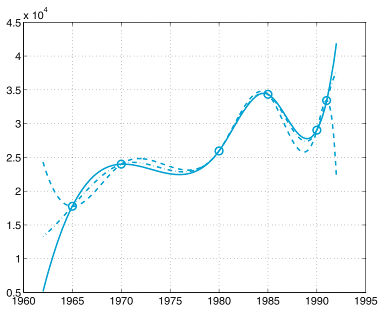
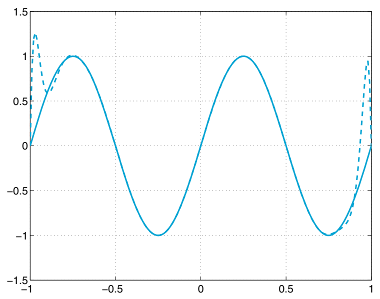



Figure 10.3. At left: comparison among the cubic spline for the data of Exercise 3.7: `s3` (*solid line*), `s3d` (*dashed line*) and `s3n` (*dotted line*). The circles denote the values used in the interpolation. At right: the interpolating polynomial (*dashed line*) and the interpolatory cubic spline (*solid line*) corresponding to the perturbed data (Solution 3.8). Note the severe oscillations of the interpolating polynomial near the end-points of the interval

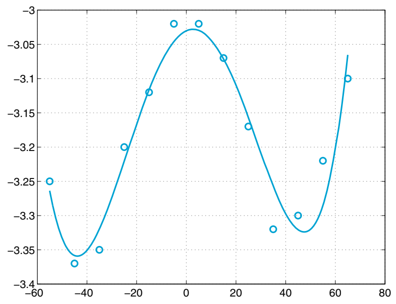
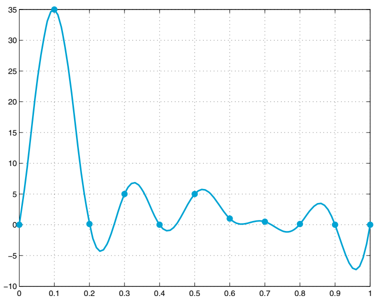



Figure 10.4. At left: least-squares polynomial of degree 4 (*solid line*) compared with the data in the first column of Table 3.1. (Solution 3.10). At right: the trigonometric interpolant obtained using the instructions in Solution 3.14. Circles refer to the available experimental data

Exercise 3.10 The coefficients (obtained by the command `polyfit`) of the requested polynomials are (only the first 4 significant digits are shown):

$K = 0.67$, $a_4 = 7.211\ 10^{-8}$, $a_3 = -6.088\ 10^{-7}$, $a_2 = -2.988\ 10^{-4}$, $a_1 = 1.650\ 10^{-3}$, $a_0 = -3.030$;
$K = 1.5$, $a_4 = -6.492\ 10^{-8}$, $a_3 = -7.559\ 10^{-7}$, $a_2 = 3.788\ 10^{-4}$, $a_1 = 1.67310^{-3}$, $a_0 = 3.149$;
$K = 2$, $a_4 = -1.050\ 10^{-7}$, $a_3 = 7.130\ 10^{-8}$, $a_2 = 7.044\ 10^{-4}$, $a_1 = -3.828\ 10^{-4}$, $a_0 = 4.926$;
$K = 3$, $a_4 = -2.319\ 10^{-7}$, $a_3 = 7.740\ 10^{-7}$, $a_2 = 1.419\ 10^{-3}$, $a_1 = -2.574\ 10^{-3}$, $a_0 = 7.315$.

In Figure 10.4, left, we show the graph of the polynomial computed using the data in the column with $K = 0.67$ of Table 3.1.

Exercise 3.11 By repeating the first 3 instructions reported in Solution 3.7 and using the command `polyfit`, we find the following values (in 10^5 Kg): 15280.12 in 1962; 27407.10 in 1977; 32019.01 in 1992, which represent good approximations to the real ones (12380, 27403 and 32059, respectively).

Exercise 3.12 We can rewrite the coefficients of the system (3.30) in terms of mean and variance by noting that the variance can be expressed as $v = \frac{1}{n+1}\sum_{i=0}^{n} x_i^2 - M^2$. Thus the coefficients of the first equation are $(n+1)$ and M, while those of the second equation are M and $(n+1)(v+M^2)$.

Exercise 3.13 The equation of the least-squares straight line is $y = a_0 + a_1 x$, where a_0 and a_1 are the solutions of system (3.30). The first equation of (3.30) provides that the point, whose abscissa is M and ordinate is $\sum_{i=0}^{n} y_i/(n+1)$, belongs to the least-squares straight line.

Exercise 3.14 We can use the command `interpft` as follows:

```
discharge = [0 35 0.125 5 0 5 1 0.5 0.125 0];
y =interpft(discharge ,100);
```

The graph of the obtained solution is reported in Figure 10.4, right.

10.4 Chapter 4

Exercise 4.1 Using the following second-order Taylor expansions of f at the point x_0, we obtain

$$\begin{aligned} f(x_1) &= f(x_0) + hf'(x_0) + \tfrac{h^2}{2} f''(x_0) + \tfrac{h^3}{6} f'''(\xi_1),\\ f(x_2) &= f(x_0) + 2hf'(x_0) + 2h^2 f''(x_0) + \tfrac{4h^3}{3} f'''(\xi_2), \end{aligned}$$

where $\xi_1 \in (x_0, x_1)$ and $\xi_2 \in (x_0, x_2)$. Replacing these two expressions in the first formula of (4.11), yields

$$\frac{1}{2h}\left[-3f(x_0) + 4f(x_1) - f(x_2)\right] = f'(x_0) + \frac{h^2}{3}[f'''(\xi_1) - 2f'''(\xi_2)],$$

then the thesis follows for a suitable $\xi_0 \in (x_0, x_2)$. A similar procedure can be used for the formula at x_n.

Exercise 4.2 By writing the second-order Taylor expansions of $f(\overline{x} \pm h)$ around $\overline{x}$, we have

$$f(\bar{x} \pm h) = f(\bar{x}) \pm hf'(\bar{x}) + \frac{h^2}{2} f''(\bar{x}) \pm \frac{h^3}{6} f'''(\xi_\pm),$$

with $\xi_- \in (\bar{x}-h, \bar{x})$ and $\xi_+ \in (\bar{x}, \bar{x}+h)$. Subtracting these two expressions and dividing by $2h$ we obtain formula (4.10) which is a second-order approximation of $f'(\overline{x})$.

Exercise 4.3 Assuming that $f \in C^4$ and proceeding as in Solution 4.2 we obtain the following errors:

$$a. \quad -\frac{1}{4}f^{(4)}(\xi)h^3, \qquad b. \quad -\frac{1}{12}f^{(4)}(\xi)h^3, \qquad c. \quad \frac{1}{6}f^{(4)}(\xi)h^3.$$

Exercise 4.4 Using the approximation (4.9), we obtain the following values:

t (months)	0	0.5	1	1.5	2	2.5	3
δn	–	78	45	19	7	3	–
n'	–	77.91	39.16	15.36	5.91	1.99	–

By comparison with the exact values of $n'(t)$ we can conclude that the computed values are sufficiently accurate.

Exercise 4.5 The quadrature error can be bounded by

$$(b-a)^3/(24M^2) \max_{x\in[a,b]} |f''(x)|,$$

where $[a,b]$ is the integration interval and M the (unknown) number of subintervals.

The function f_1 is infinitely differentiable. From the graph of f_1'' we infer that $|f_1''(x)| \leq 2$ in the integration interval. Thus the integration error for f_1 is less than 10^{-4} provided that $2\cdot 5^3/(24M^2) < 10^{-4}$, that is $M > 322$.

Also the function f_2 is differentiable to any order. Since $\max_{x\in[0,\pi]} |f_2''(x)| = \sqrt{2}e^{3\pi/4}$, the integration error is less than 10^{-4} provided that $M > 439$. These inequalities actually provide an over estimation of the integration errors. Indeed, the (effective) minimum number of intervals which ensures that the error is below the fixed tolerance of 10^{-4} is much lower than that predicted by our result (for instance, for the function f_1 this number is 71). Finally, we note that, since f_3 is not differentiable at both $x = 0$ and $x = 1$, the theoretical error estimate doesn't hold.

Exercise 4.6 On each interval I_k, $k = 1,\ldots,M$, the error is equal to $H^3/24 f''(\xi_k)$ with $\xi_k \in [x_{k-1}, x_k]$ and hence the global error will be $H^3/24 \sum_{k=1}^{M} f''(\xi_k)$. Since f'' is a continuous function in $[a,b]$ there exists a point $\xi \in [a,b]$ such that $f''(\xi) = \frac{1}{M}\sum_{k=1}^{M} f''(\xi_k)$. Using this result and the fact that $MH = b - a$, we derive equation (4.14).

Exercise 4.7 This effect is due to the accumulation of local errors on each sub-interval.

Exercise 4.8 By construction, the mid-point formula integrates exactly the constants. To verify that the linear polynomials also are exactly integrated, it is sufficient to verify that $I(x) = I_{PM}(x)$. As a matter of fact we have

$$I(x) = \int_a^b x\, dx = \frac{b^2 - a^2}{2}, \qquad I_{PM}(x) = (b-a)\frac{b+a}{2}.$$

Exercise 4.9 For the function f_1 we find $M = 71$ if we use the trapezoidal formula and only $M = 8$ for the composite Gauss-Legendre formula with $n = 1$. (For this formula we can use Program 10.1.) Indeed, the computational advantage of this latter formula is evident.

Program 10.1. gausslegendre: Gauss-Legendre composite quadrature formula, with $n = 1$

```
function intGL=gausslegendre(a,b,f,M,varargin)
y = [-1/sqrt(3),1/sqrt(3)];
H2 = (b-a)/(2*M);
z = [a:2*H2:b];
zM = (z(1:end-1)+z(2:end))*0.5;
x = [zM+H2*y(1), zM+H2*y(2)];
f = f(x,varargin{:});
intGL = H2*sum(f);
return
```

Exercise 4.10 Equation (4.18) states that the quadrature error for the composite trapezoidal formula with $H = H_1$ is equal to CH_1^2, with $C = -\frac{b-a}{12}f''(\xi)$. If f'' does not vary "too much", we can assume that also the error with $H = H_2$ behaves like CH_2^2. Then, by equating the two expressions

$$I(f) \simeq I_1 + CH_1^2, \quad I(f) \simeq I_2 + CH_2^2, \tag{10.3}$$

we obtain $C = (I_1 - I_2)/(H_2^2 - H_1^2)$. Using this value in one of the expressions (10.3), we obtain equation (4.35), that is, a better approximation than the one produced by I_1 or I_2.

Exercise 4.11 We seek the maximum positive integer p such that $I_{appr}(x^p) = I(x^p)$. For $p = 0, 1, 2, 3$ we find the following nonlinear system with 4 equations in the 4 unknowns α, β, $\bar{x}$ and $\bar{z}$:

$$\begin{aligned}
p = 0 &\rightarrow \alpha + \beta = b - a,\\
p = 1 &\rightarrow \alpha\bar{x} + \beta\bar{z} = \frac{b^2 - a^2}{2},\\
p = 2 &\rightarrow \alpha\bar{x}^2 + \beta\bar{z}^2 = \frac{b^3 - a^3}{3},\\
p = 3 &\rightarrow \alpha\bar{x}^3 + \beta\bar{z}^3 = \frac{b^4 - a^4}{4}.
\end{aligned}$$

From the first two equations we can eliminate α and $\bar{z}$ and reduce the system to a new one in the unknowns β and $\bar{x}$. In particular, we find a second-order equation in β from which we can compute β as a function of $\bar{x}$. Finally, the nonlinear equation in $\bar{x}$ can be solved by the Newton method, yielding two values of $\bar{x}$ that are the nodes of the Gauss-Legendre quadrature formula with $n = 1$.

Exercise 4.12 Since

$$f_1^{(4)}(x) = 24\frac{1-10(x-\pi)^2+5(x-\pi)^4}{(1+(x-\pi)^2)^5},$$
$$f_2^{(4)}(x) = -4e^x\cos(x),$$

we find that the maximum of $|f_1^{(4)}(x)|$ is bounded by $M_1 \simeq 23$, while that of $|f_2^{(4)}(x)|$ by $M_2 \simeq 18$. Consequently, from (4.22) we obtain $H < 0.21$ in the first case and $H < 0.16$ in the second case.

Exercise 4.13 The MATLAB commands:

```
syms x
I=int(exp(-x^2/2),0,2);
Iex=eval(I)
```

yields the value 1.19628801332261 for the integral at hand.

The Gauss-Legendre formula applied to the same interval with $M = 1$ would provide the value 1.20278027622354 (with an absolute error equal to 6.4923e-03), while the simple Simpson formula gives 1.18715264069572 with a slightly larger error, equal to 9.1354e-03.

Exercise 4.14 We note that $I_k > 0$ $\forall k$, since the integrand is non-negative. Therefore, we expect that all the values produced by the recursive formula should be non-negative. Unfortunately, the recursive formula is unstable to the propagation of roundoff errors and produces negative elements:

```
I(1)=1/exp(1); for k=2:20, I(k)=1-k*I(k-1); end
```

The result is `I(20) = 104.86` in MATLAB, while Octave produces `I(20) = -30.1924`. Using the composite Simpson formula, with $M \geq 16$, we can compute the integral with the desired accuracy, as a matter of fact, denoting by $f(x)$ the integrand function, the absolute value of its fourth derivative is bounded by $M \simeq 1.46\ 10^5$. Consequently, from (4.22) we obtain $H < 0.066$.

Exercise 4.15 The idea of Richardson's extrapolation is general and can be applied to any quadrature formula. By proceeding as in Solution 4.10, recalling that both Simpson and Gauss quadrature formulas are fourth-order accurate, formula (4.35) reads

$$I_R = I_1 + (I_1 - I_2)/(H_2^4/H_1^4 - 1).$$

For the Simpson formula we obtain

$$I_1 = 1.19616568040561,\ I_2 = 1.19628173356793,\ \Rightarrow\ I_R = 1.19628947044542,$$

with an absolute error $I(f) - I_R = -1.4571e - 06$ (we gain two orders of magnitude with respect to I_1 and a factor 1/4 with respect to I_2). Using the Gauss-Legendre formula we obtain (the errors are reported between parentheses):

$$\begin{aligned} I_1 &= 1.19637085545393\ \ (-8.2842e-05),\\ I_2 &= 1.19629221796844\ \ (-4.2046e-06),\\ I_R &= 1.19628697546941\ \ (1.0379e-06). \end{aligned}$$

The advantage of using the Richardson extrapolation method is evident.

Exercise 4.16 We must compute by the Simpson formula the values $j(r,0) = \sigma/(\varepsilon_0 r^2)\int_0^r f(\xi)d\xi$ with $r = k/10$, for $k = 1,\dots,10$ and $f(\xi) = e^{\xi}\xi^2$.

In order to estimate the integration error we need the fourth derivative $f^{(4)}(\xi) = e^{\xi}(\xi^2 + 8\xi + 12)$. The maximum of $f^{(4)}$ in the integration interval $[0, r]$ is attained at $\xi = r$ since $f^{(4)}$ is monotonically increasing. For a given r the error is below 10^{-10} provided that $H^4 < 10^{-10}2880/(rf^{(4)}(r))$. For $r = k/10$ with $k = 1,\dots,10$ by the following instructions we can compute the minimum numbers of subintervals which ensure that the previous inequalities are satisfied:

```
r=[0.1:0.1:1]; maxf4=exp(r).*(r.^2+8*r+12);
H=(10^(-10)*2880./(r.*maxf4)).^(1/4); M=fix(r./H)

M =
       4      11      20      30      41      53      67      83     100
118
```

Therefore, the values of $j(r,0)$ are computed by running the following instructions:

```
sigma=0.36; epsilon0 = 8.859e-12;
f=@(x) exp(x).*x.^2;
for k = 1:10
   r = k/10;
   j(k)=simpsonc(0,r,M(k),f);
   j(k) = j(k)*sigma/(r^2*epsilon0);
end
```

Exercise 4.17 We compute $E(213)$ using the Simpson composite formula by increasing the number of intervals until the difference between two consecutive approximations (divided by the last computed value) is less than 10^{-11}:

```
f=@(x) 1./(x.^5.*(exp(1.432./(213*x))-1));
a=3.e-04; b=14.e-04;
i=1; err = 1; Iold = 0; while err >= 1.e-11
I=2.39e-11*simpsonc(a,b,i,f);
err = abs(I-Iold)/abs(I);
Iold=I;
i=i+1;
end
```

The procedure returns the value $i = 59$. Therefore, using 58 equispaced intervals we can compute the integral $E(213)$ with ten exact significant digits. The same result could be obtained by the Gauss-Legendre formula using 53 intervals. Note that as many as 1609 intervals would be nedeed if using the composite trapezoidal formula.

Exercise 4.18 On the whole interval the given function is not regular enough to allow the application of the theoretical convergence result (4.22). One possibility is to decompose the integral into the sum of two intervals, $[0, 0.5]$ and $[0.5, 1]$, in which the function is regular (it is actually a polynomial of degree 2 in each sub-interval). In particular, if we use the Simpson rule on each interval we can even integrate f exactly.

10.5 Chapter 5

Exercise 5.1 Let x_n denote the number of algebraic operations (sums, subtractions and multiplications) required to compute one determinant of a matrix of order $n \times n$ by the Laplace rule (1.8). The following recursive formula holds

$$x_k - kx_{k-1} = 2k - 1, \qquad k \geq 2,$$

with $x_1 = 0$. Multiplying both sides of this equation by $1/k!$, we obtain

$$\frac{x_k}{k!} - \frac{x_{k-1}}{(k-1)!} = \frac{2k-1}{k!}$$

and summing both sides from 2 to n gives the solution:

$$x_n = n! \sum_{k=2}^{n} \frac{2k-1}{k!}.$$

Recalling that $\sum_{k=0}^{\infty} \frac{1}{k!} = e$, it holds

$$\sum_{k=2}^{n} \frac{2k-1}{k!} = 2\sum_{k=1}^{n-1} \frac{1}{k!} - \sum_{k=2}^{n} \frac{1}{k!} \simeq 2.718,$$

whence $x_n \simeq 3n!$. It is worth mentioning that the Cramer rule (see Section 5.2) requires about $3(n+1)!$ operations to solve a square linear system of order n with full matrix.

Exercise 5.2 We use the following MATLAB commands to compute the determinants and the corresponding CPU-times:

```
t = []; NN=3:500;
for n = NN
A=magic(n); tt=cputime; d=det(A); t=[t, cputime-tt];
end
```

Let us compute the coefficients of the cubic least-squares polynomial that approximate the data `NN=[3:500]` and `t`

```
c=polyfit(NN,t,3)
c =
    1.4055e-10    7.1570e-08   -3.6686e-06    3.1897e-04
```

If we compute the fourth degree least-squares polynomial

```
c=polyfit(NN,t,4)
```

we obtain the following coefficients:

```
c =
   7.6406e-15    1.3286e-10    7.4064e-08   -3.9505e-06    3.2637e-04
```

that is, the coefficient of n^4 is close to the machine precision while the other ones are quite unchanged with respect to the projection on $\mathbb{P}_3$. From this result, we can conclude that in MATLAB the CPU-time required for computing the determinant of a matrix of dimension `n` scales as `n`^{3}.

Exercise 5.3 Denoting by A_i the principal submatrix of A of order i, we have: $\det\mathrm{A}_1 = 1$, $\det\mathrm{A}_2 = \varepsilon$, $\det\mathrm{A}_3 = \det\mathrm{A} = 2\varepsilon + 12$. Consequently, if $\varepsilon = 0$ the second principal submatrix is singular and the LU factorization of A does not exist (see Proposition 5.1). The matrix A is singular if $\varepsilon = -6$. In this case the LU factorization exists and yields

$$\mathrm{L} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & 1.25 & 1 \end{bmatrix}, \quad \mathrm{U} = \begin{bmatrix} 1 & 7 & 3 \\ 0 & -12 & -4 \\ 0 & 0 & 0 \end{bmatrix}.$$

Nevertheless, note that U is singular (as we could have predicted since A is singular) and the upper triangular system $\mathrm{U}\mathbf{x} = \mathbf{y}$ admits infinite solutions. We notice that the backward substitutions (5.10) cannot be applied because of the same reason.

Exercise 5.4 Let us consider algorithm 5.13. At step $k = 1$, $n-1$ divisions were used to calculate the l_{i1} entries for $i = 2, \ldots, n$. Then $(n-1)^2$ multiplications and $(n-1)^2$ additions were used to create the new entries $a_{ij}^{(2)}$, for $i, j = 2, \ldots, n$. At step $k = 2$, the number of divisions is $(n-2)$, while the number of multiplications and additions will be $(n-2)^2$. At final step $k = n-1$ only 1 addition, 1 multiplication and 1 division is required. Thus, using the identies

$$\sum_{s=1}^{q} s = \frac{q(q+1)}{2}, \qquad \sum_{s=1}^{q} s^2 = \frac{q(q+1)(2q+1)}{6}, \quad q \geq 1,$$

we can conclude that to complete the LU factorization we need the following number of operations

$$\begin{aligned} \sum_{k=1}^{n-1} \sum_{i=k+1}^{n} \left(1 + \sum_{j=k+1}^{n} 2\right) &= \sum_{k=1}^{n-1} (n-k)(1 + 2(n-k)) \\ = \sum_{j=1}^{n-1} j + 2\sum_{j=1}^{n-1} j^2 = \frac{(n-1)n}{2} + 2\frac{(n-1)n(2n-1)}{6} &= \frac{2}{3}n^3 - \frac{n^2}{2} - \frac{n}{6}. \end{aligned}$$

Exercise 5.5 By definition, the inverse X of a matrix $\mathrm{A} \in \mathbb{R}^{n\times n}$ satisfies $\mathrm{XA} = \mathrm{AX} = \mathrm{I}$. Therefore, for $j = 1, \ldots, n$ the column vector $\mathbf{x}_j$ of X is the solution of the linear system $\mathrm{A}\mathbf{x}_j = \mathbf{e}_j$, where $\mathbf{e}_j$ is the jth vector of the canonical basis of $\mathbb{R}^n$ with all components equal to zero except the jth that is equal to 1. After computing the LU factorization of A, the computation of the inverse of A requires the solution of n linear systems with the same matrix and different right-hand sides.

Exercise 5.6 Using the Program 5.1 we compute the L and U factors:

$$\mathrm{L} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & -3.38 \cdot 10^{15} & 1 \end{bmatrix}, \quad \mathrm{U} = \begin{bmatrix} 1 & 1 & 3 \\ 0 & -8.88 \cdot 10^{-16} & 14 \\ 0 & 0 & 4.73 \cdot 10^{-16} \end{bmatrix}.$$

If we compute their product we obtain the matrix

```
L*U
ans =
    1.0000    1.0000     3.0000
    2.0000    2.0000    20.0000
    3.0000    6.0000     0.0000
```

which differs from A since the entry in position (3,3) is equal to 0 while in A it is equal to 4.
The accurate computation of both L and U can be accomplished by invoking a partial pivoting by rows, indeed by the instruction `[L,U,P]=lu(A)` we obtain the correct results.

Exercise 5.7 Usually, only the triangular (upper or lower) part of a symmetric matrix is stored. Therefore, any operation that does not respect the symmetry of the matrix is not optimal in view of the memory storage. This is the case when row pivoting is carried out. A possibility is to exchange simultaneously rows and columns having the same index, limiting therefore the choice of the pivot only to the diagonal elements. More generally, a pivoting strategy involving exchange of rows and columns is called *complete pivoting* (see, e.g., [QSS07, Chap. 3]).

Exercise 5.8 The symbolic computation of the L and U factors yields

$$\mathrm{L} = \begin{bmatrix} 1 & 0 & 0 \\ (\varepsilon-2)/2 & 1 & 0 \\ 0 & -1/\varepsilon & 1 \end{bmatrix}, \quad \mathrm{U} = \begin{bmatrix} 2 & -2 & 0 \\ 0 & \varepsilon & 0 \\ 0 & 0 & 3 \end{bmatrix},$$

thus $l_{32} \to \infty$, when $\varepsilon \to 0$. If we choose $\mathbf{b} = (0, \varepsilon, 2)^T$, it is easy to verify that $\mathbf{x} = (1,1,1)^T$ is the exact solution of $\mathrm{A}\mathbf{x} = \mathbf{b}$. To analyze the error with respect to the exact solution for $\varepsilon \to 0$, let us take $\varepsilon = 10^{-k}$, for $k = 0, \ldots, 9$. The following instructions

```
e=1;xex=ones(3,1);err=[];
for k=1:10
b=[0;e;2];
L=[1 0 0; (e-2)*0.5 1 0; 0 -1/e 1];
U=[2 -2 0; 0 e 0; 0 0 3];
y=L\b; x=U\y;
err(k)=norm(x-xex)/norm(xex); e=e*0.1;
end
```

yield

```
   err =
       0    0    0    0    0    0    0    0    0    0
```

i.e., the numerical solution is not affected by rounding errors. This can be explained by noticing that all the entries of L, U and **b** are floating point numbers not affected by rounding errors and, unusually, no rounding errors are propagated during both forward and backward substitutions, even if the condition number of A is proportional to $1/\varepsilon$.
On the contrary, by setting $\mathbf{b} = (2\log(2.5) - 2, (\varepsilon - 2)\log(2.5) + 2, 2)^T$, which corresponds to the exact solution $\mathbf{x} = (\log(2.5), 1, 1)^T$, and analyzing the relative error for $\varepsilon = 1/3 \cdot 10^{-k}$, for $k = 0, \ldots, 9$, the instructions

```
e=1/3; xex=[log(5/2),1,1]'; err=[];
for k=1:10
b=[2*log(5/2)-2,(e-2)*log(5/2)+2,2]';
L=[1 0 0; (e-2)*0.5 1 0; 0 -1/e 1];
U=[2 -2 0; 0 e 0; 0 0 3];
y=L\b; x=U\y;
err(k)=norm(x-xex)/norm(xex); e=e*0.1;
end
```

provide

```
err =
  Columns 1 through 5
   1.8635e-16   5.5327e-15   2.6995e-14   9.5058e-14   1.3408e-12
  Columns 6 through 10
   1.2828e-11   4.8726e-11   4.5719e-09   4.2624e-08   2.8673e-07
```

In the latter case the error depends on the condition number of A, which obeys the law $K(A) = C/\varepsilon$ and satisfies the estimate (5.34).

Exercise 5.9 The computed solutions become less and less accurate when i increases. Indeed, the error norms are equal to $1.10 \cdot 10^{-14}$ for $i = 1$, to $9.32 \cdot 10^{-10}$ for $i = 2$ and to $2.51 \cdot 10^{-7}$ for $i = 3$. (We warn the reader that these results indeed change depending upon the different MATLAB versions used!!) This can be explained by observing that the condition number of A_i increases as i increases. Indeed, using the command `cond` we find that the condition number of A_i is $\simeq 10^3$ for $i = 1$, $\simeq 10^7$ for $i = 2$ and $\simeq 10^{11}$ for $i = 3$.

Exercise 5.10 If $(\lambda, \mathbf{v})$ are an eigenvalue-eigenvector pair of a matrix A, then λ^2 is an eigenvalue of A^2 with the same eigenvector. Indeed, from $\mathrm{A}\mathbf{v} = \lambda\mathbf{v}$ follows $\mathrm{A}^2\mathbf{v} = \lambda\mathrm{A}\mathbf{v} = \lambda^2\mathbf{v}$. Consequently, if A is symmetric and positive definite $K(\mathrm{A}^2) = (K(\mathrm{A}))^2$.

Exercise 5.11 The iteration matrix of the Jacobi method is:

$$\mathrm{B}_J = \begin{bmatrix} 0 & 0 & -\alpha^{-1} \\ 0 & 0 & 0 \\ -\alpha^{-1} & 0 & 0 \end{bmatrix}.$$

Its eigenvalues are $\{0, \alpha^{-1}, -\alpha^{-1}\}$. Thus the method converges if $|\alpha| > 1$.

The iteration matrix of the Gauss-Seidel method is

$$\mathrm{B}_{GS} = \begin{bmatrix} 0 & 0 & -\alpha^{-1} \\ 0 & 0 & 0 \\ 0 & 0 & \alpha^{-2} \end{bmatrix}$$

with eigenvalues $\{0, 0, \alpha^{-2}\}$. Therefore, the method converges if $|\alpha| > 1$. In particular, since $\rho(\mathrm{B}_{GS}) = [\rho(\mathrm{B}_J)]^2$, the Gauss-Seidel converges more rapidly than the Jacobi method.

Exercise 5.12 A sufficient condition for the convergence of the Jacobi and the Gauss-Seidel methods is that A is strictly diagonally dominant. The second row of A satisfies the condition of diagonal dominance provided that $|\beta| < 5$. Note that if we require directly that the spectral radii of the iteration matrices are less than 1 (which is a sufficient and necessary condition for convergence), we find the (less restrictive) limitation $|\beta| < 25$ for both methods.

Exercise 5.13 The relaxation method in vector form is

$$(\mathrm{I} - \omega \mathrm{D}^{-1}\mathrm{E})\mathbf{x}^{(k+1)} = [(1-\omega)\mathrm{I} + \omega \mathrm{D}^{-1}\mathrm{F}]\mathbf{x}^{(k)} + \omega \mathrm{D}^{-1}\mathbf{b}$$

where $\mathrm{A} = \mathrm{D} - (\mathrm{E} + \mathrm{F})$, D being the diagonal of A, and -E and -F the lower (resp. upper) part of A. The corresponding iteration matrix is

$$\mathrm{B}(\omega) = (\mathrm{I} - \omega \mathrm{D}^{-1}\mathrm{E})^{-1}[(1-\omega)\mathrm{I} + \omega \mathrm{D}^{-1}\mathrm{F}].$$

If we denote by λ_i the eigenvalues of $\mathrm{B}(\omega)$, we obtain

$$\begin{aligned}\left|\prod_{i=1}^{n}\lambda_i\right| &= |\det \mathrm{B}(\omega)| \\ &= |\det[(I-\omega D^{-1}E)^{-1}]| \cdot |\det[(1-\omega)I + \omega D^{-1}F)]|.\end{aligned}$$

Noticing that, given two matrices A and B with $\mathrm{A} = \mathrm{I} + \alpha \mathrm{B}$, for any $\alpha \in \mathbb{R}$ it holds $\lambda_i(\mathrm{A}) = 1 + \alpha\lambda_i(\mathrm{B})$, and that all the eigenvalues of both $\mathrm{D}^{-1}E$ and $\mathrm{D}^{-1}F$ are null, we have

$$\left|\prod_{i=1}^{n}\lambda_i\right| = \left|\prod_{i=1}^{n}\frac{(1-\omega)+\omega\lambda_i(\mathrm{D}^{-1}F)}{1-\omega\lambda_i(\mathrm{D}^{-1}E)}\right| = |1-\omega|^n.$$

Therefore, at least one eigenvalue must satisfy the inequality $|\lambda_i| \geq |1-\omega|$. Thus, a necessary condition to ensure convergence is that $|1-\omega| < 1$, that is, $0 < \omega < 2$.

Exercise 5.14 Matrix $\mathrm{A} = \begin{bmatrix} 3 & 2 \\ 2 & 6 \end{bmatrix}$ is strictly diagonally dominant by rows, a sufficient condition for the Gauss-Seidel method to converge. On the contrary, matrix $\mathrm{A} = \begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix}$ is not strictly diagonally dominant by rows, however it is symmetric. Moreover, we can easily verify that it is positive definite, i.e. $\mathbf{z}^T\mathrm{A}\mathbf{z} > 0$ for any $\mathbf{z} \neq \mathbf{0}$ of $\mathbb{R}^2$. We perform the following computations by MATLAB (obviously, for this simple case, we could perform them by hands!):

```
syms z1 z2 real
z=[z1;z2]; A=[1 1; 1 2];
pos=z'*A*z; simple(pos)
ans =
   z1^2+2*z1*z2+2*z2^2
ans =
   z1^2+2*z1*z2+2*z2^2
```

where the command `syms z1 z2 real` converts the variables `z1` and `z2` from symbolic to real type, while the command `simple` tries several algebraic simplifications of `pos` and returns the shortest. It is easy to see that the computed quantity is positive since it can be rewritten as `(z1+z2)^2+z2^2`. Thus, the given matrix is symmetric and positive definite, a sufficient condition for the Gauss-Seidel method to converge.

Exercise 5.15 We find:

for the Jacobi method: $\begin{cases} x_1^{(1)} &= \frac{1}{2}(1 - x_2^{(0)}), \\ x_2^{(1)} &= -\frac{1}{3}(x_1^{(0)}); \end{cases} \Rightarrow \begin{cases} x_1^{(1)} &= \frac{1}{4}, \\ x_2^{(1)} &= -\frac{1}{3}; \end{cases}$

for the Gauss-Seidel method: $\begin{cases} x_1^{(1)} &= \frac{1}{2}(1 - x_2^{(0)}), \\ x_2^{(1)} &= -\frac{1}{3}x_1^{(1)}, \end{cases} \Rightarrow \begin{cases} x_1^{(1)} &= \frac{1}{4}, \\ x_2^{(1)} &= -\frac{1}{12}\,; \end{cases}$

for the gradient method, we first compute the initial residual

$$\mathbf{r}^{(0)} = \mathbf{b} - \mathrm{A}\mathbf{x}^{(0)} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} - \begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix} \mathbf{x}^{(0)} = \begin{bmatrix} -3/2 \\ -5/2 \end{bmatrix}.$$

Then, since

$$\mathrm{P}^{-1} = \begin{bmatrix} 1/2 & 0 \\ 0 & 1/3 \end{bmatrix},$$

we have $\mathbf{z}^{(0)} = \mathrm{P}^{-1}\mathbf{r}^{(0)} = (-3/4, -5/6)^T$. Therefore

$$\alpha_0 = \frac{(\mathbf{z}^{(0)})^T\mathbf{r}^{(0)}}{(\mathbf{z}^{(0)})^T\mathrm{A}\mathbf{z}^{(0)}} = \frac{77}{107},$$

and

$$\mathbf{x}^{(1)} = \mathbf{x}^{(0)} + \alpha_0\mathbf{z}^{(0)} = (197/428, -32/321)^T.$$

Exercise 5.16 In the stationary case, the eigenvalues of the matrix $\mathrm{B}_\alpha = \mathrm{I} - \alpha\mathrm{P}^{-1}\mathrm{A}$ are $\mu_i(\alpha) = 1 - \alpha\lambda_i$, λ_i being the ith eigenvalue of $\mathrm{P}^{-1}\mathrm{A}$. Then

$$\rho(\mathrm{B}_\alpha) = \max_{i=1,\ldots,n} |1 - \alpha\lambda_i| = \max\{|1 - \alpha\lambda_{min}|, |1 - \alpha\lambda_{max}|\}.$$

Thus, the optimal value of α (that is the value that minimizes the spectral radius of the iteration matrix) is the root of the equation

$$1 - \alpha\lambda_{min} = \alpha\lambda_{max} - 1$$

which yields (5.58). Formula (5.72) follows now by a direct computation of $\rho(\mathrm{B}_{\alpha_{opt}})$.

Exercise 5.17 We have to minimize the function $\Phi(\alpha) = \|\mathbf{e}^{(k+1)}\|_{\mathrm{A}}^2$ with respect to $\alpha \in \mathbb{R}$. Since $\mathbf{e}^{(k+1)} = \mathbf{x} - \mathbf{x}^{(k+1)} = \mathbf{e}^{(k)} - \alpha\mathbf{z}^{(k)}$, we obtain

$$\Phi(\alpha) = \|\mathbf{e}^{(k+1)}\|_{\mathrm{A}}^2 = \|\mathbf{e}^{(k)}\|_{\mathrm{A}}^2 + \alpha^2\|\mathbf{z}^{(k)}\|_{\mathrm{A}}^2 - 2\alpha(\mathbf{z}^{(k)})^T\mathrm{A}\mathbf{e}^{(k)}.$$

The minimum of $\Phi(\alpha)$ is found in correspondence to the value α_k such that $\Phi'(\alpha_k) = 0$, i.e.,

$$\alpha_k\|\mathbf{z}^{(k)}\|_{\mathrm{A}}^2 - (\mathbf{z}^{(k)})^T\mathrm{A}\mathbf{e}^{(k)} = 0,$$

so that $\alpha_k = ((\mathbf{z}^{(k)})^T\mathrm{A}\mathbf{e}^{(k)})/\|\mathbf{z}^{(k)}\|_{\mathrm{A}}^2$. Finally, (5.60) follows by noticing that $\mathrm{A}\mathbf{e}^{(k)} = \mathbf{r}^{(k)}$.

Exercise 5.18 We provide two possible proofs.
1. Note that $\mathrm{P}^{-1}\mathrm{A} = \mathrm{P}^{-1/2}(\mathrm{P}^{-1/2}\mathrm{A}\mathrm{P}^{-1/2})\mathrm{P}^{1/2}$ where $\mathrm{P}^{1/2}$ is the square root of P (see, e.g. [QV94, Sect. 2.5]). Since P is symmetric positive definite, $\mathrm{P}^{1/2}$ is symmetric and positive definite and it is the unique solution of the matrix equation $\mathrm{X}^2 = \mathrm{P}$. This shows that $\mathrm{P}^{-1}\mathrm{A}$ is similar to the matrix $\mathrm{P}^{-1/2}\mathrm{A}\mathrm{P}^{-1/2}$ which is symmetric positive definite.
2. The eigenpairs $(\mu, \mathbf{y})$ of $\mathrm{P}^{-1}\mathrm{A}$ satisfy the equation $\mathrm{P}^{-1}\mathrm{A}\mathbf{y} = \mu\mathbf{y}$, that is $\mathrm{A}\mathbf{y} = \mu\mathrm{P}\mathbf{y}$, therefore $\mu = (\mathbf{y}^T A\mathbf{y})/(\mathbf{y}^T\mathrm{P}\mathbf{y}) > 0$ since both A and P are symmetric positive definite.

Exercise 5.19 The matrix associated to the Leontieff model is symmetric, but not positive definite. Indeed, using the following instructions:

```
for i=1:20;
  for j=1:20;
    C(i,j)=i+j;
  end;
end;
A=eye(20)-C;
[min(eig(A)), max(eig(A))]

ans =
 -448.58   30.583
```

we can see that the minimum eigenvalue is a negative number and the maximum eigenvalue is a positive number. Therefore, the convergence of the gradient method is not guaranteed. However, since A is nonsingular, the given system is equivalent to the system $\mathrm{A}^T\mathrm{A}\mathbf{x} = \mathrm{A}^T\mathbf{b}$, where $\mathrm{A}^T\mathrm{A}$ is symmetric and positive definite. We solve the latter by the gradient method requiring that the norm of the residual be less than 10^{-10} and starting from the initial data $\mathbf{x}^{(0)} = \mathbf{0}$:

```
b = [1:20]';  AA=A'*A; b=A'*b; x0 = zeros(20,1);
[x,iter]=itermeth(AA,b,x0,100,1.e-10);
```

The method converges in 15 iterations. A drawback of this approach is that the condition number of the matrix $\mathrm{A}^T\mathrm{A}$ is, in general, larger than the condition number of A.

10.6 Chapter 6

Exercise 6.1 A_1: the power method converges in 34 iterations to the value 2.00000000004989. A_2: starting from the same initial vector, the power method requires now 457 iterations to converge to the value 1.99999999990611. The slower convergence rate can be explained by observing that the two largest eigenvalues are very close one another. Finally, for the matrix A_3 the method doesn't converge since A_3 features two distinct eigenvalues (i and $-i$) of maximum modulus.

Exercise 6.2 The Leslie matrix associated with the values in the table is

$$A = \begin{bmatrix} 0 & 0.5 & 0.8 & 0.3 \\ 0.2 & 0 & 0 & 0 \\ 0 & 0.4 & 0 & 0 \\ 0 & 0 & 0.8 & 0 \end{bmatrix}.$$

Using the power method we find $\lambda_1 \simeq 0.5353$. The normalized distribution of this population for different age intervals is given by the components of the corresponding unitary eigenvector, that is, $\mathbf{x}_1 \simeq (0.8477, 0.3167, 0.2367, 0.3537)^T$.

Exercise 6.3 We rewrite the initial guess as

$$\mathbf{y}^{(0)} = \beta^{(0)} \left(\alpha_1 \mathbf{x}_1 + \alpha_2 \mathbf{x}_2 + \sum_{i=3}^{n} \alpha_i \mathbf{x}_i \right),$$

with $\beta^{(0)} = 1/\|\mathbf{x}^{(0)}\|$. By calculations similar to those carried out in Section 6.2, at the generic step k we find:

$$\mathbf{y}^{(k)} = \gamma^k \beta^{(k)} \left(\alpha_1 \mathbf{x}_1 e^{ik\vartheta} + \alpha_2 \mathbf{x}_2 e^{-ik\vartheta} + \sum_{i=3}^{n} \alpha_i \frac{\lambda_i^k}{\gamma^k} \mathbf{x}_i \right).$$

Therefore, when $k \to \infty$, the first two terms don't vanish and, due to the opposite sign of the exponents, the sequence of the $\mathbf{y}^{(k)}$ oscillates and cannot converge.

Exercise 6.4 If A is non-singular, from the eigenvalue equation $A\mathbf{x} = \lambda\mathbf{x}$, we deduce $A^{-1}A\mathbf{x} = \lambda A^{-1}\mathbf{x}$, and therefore $A^{-1}\mathbf{x} = (1/\lambda)\mathbf{x}$.

Exercise 6.5 The power method applied to the matrix A generates an oscillating sequence of approximations of the maximum modulus eigenvalue (see, Figure 10.5). This behavior is due to the fact that the matrix A has two distinct eigenvalues of maximum modulus.

Exercise 6.6 Since the eigenvalues of a real symmetric matrix are all real, they lie inside a closed bounded interval $[\lambda_a, \lambda_b]$. Our aim is to estimate both λ_a and λ_b. To compute the eigenvalue of maximum modulus of A we use Program 6.1:

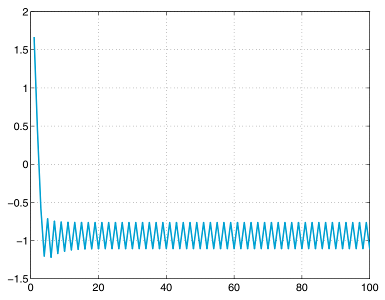


Figure 10.5. The approximations of the maximum modulus eigenvalue of the matrix of Solution 6.5 computed by the power method

```
A=wilkinson(7);
x0=ones(7,1); tol=1.e-15; nmax=100;
[lambdab,x,iter]=eigpower(A,tol,nmax,x0);
```

After 35 iterations we obtain `lambdab=3.76155718183189`. Since λ_a is the eigenvalue of A farest from λ_b, in order to compute it we apply the power method to the matrix $\mathrm{A}_b = \mathrm{A} - \lambda_b \mathrm{I}$, that is we compute the maximum modulus eigenvalue of the matrix A_b. Then we will set $\lambda_a = \lambda + \lambda_b$. The instructions

```
[lambda,x,iter]=eigpower(A-lambdab*eye(7),tol,nmax,x0);
lambdaa=lambda+lambdab
```

yield `lambdaa =-1.12488541976457` after 33 iterations. These results are satisfactory approximations of the extremal eigenvalues of A.

Exercise 6.7 Let us start by considering the matrix A. We observe that there is an isolated row circle centered at $(9, 0)$ with radius equal to 1, that can contain only one eigenvalue (say λ_1), in view of Proposition 6.1. Therefore $\lambda_1 \in \mathbb{R}$, more precisely $\lambda_1 \in (8, 10)$. Moreover, from Figure 10.6, right, we note that A features two other isolated column circles centered at $(2, 0)$ and $(4, 0)$, respectively, both with radius equal to 1/2. Therefore A has two other real eigenvalues $\lambda_2 \in (1.5, 2.5)$ and $\lambda_3 \in (3.4, 4.5)$. Since all the coefficients of A are real, we can conclude that also the fourth eigenvalue will be real.

Let us consider now the matrix B that admits only one isolated column circle (see Figure 10.7 right), centered at $(-5, 0)$ and with radius 1/2. Then, thanks to the previous consideration the corresponding eigenvalue must be real and it will belong to the interval $(-5.5, -4.5)$. The remaining eigenvalues can be either all real, or one real and 2 complex.

Exercise 6.8 The row circles of A feature an isolated circle of center (5,0) and radius 2 the maximum modulus eigenvalue must belong to. Therefore, we can set the value of the shift equal to 5. The comparison between the number of iterations and the computational cost of the power method with and without shift can be found using the following commands:

```
A=[5 0 1 -1; 0 2 0 -1/2; 0 1 -1 1; -1 -1 0 0];
```

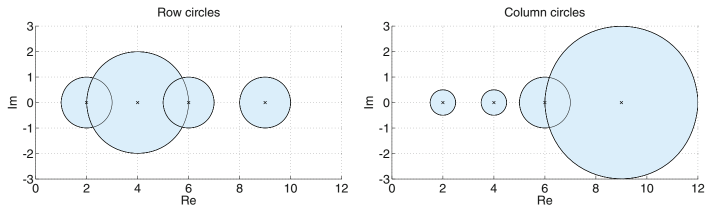


Figure 10.6. Row circles (at left) and column circles (at right) of the matrix A of Solution 6.7

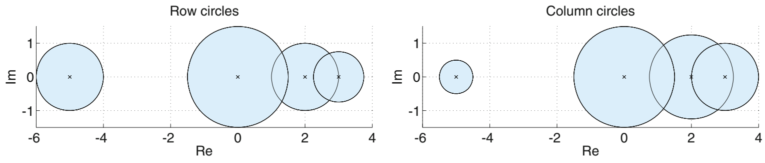


Figure 10.7. Row circles (at left) and column circles (at right) circles of the matrix B of Solution 6.7

```
tol=1e-14; x0=[1 2 3 4]';  nmax=1000;
tic; [lambda,x,iter]=eigpower(A,tol,nmax,x0);
toc, iter

Elapsed time is 0.001854 seconds.
iter = 35

tic; [lambda,x,iter]=invshift(A,5,tol,nmax,x0);
toc, iter

Elapsed time is 0.000865 seconds.
iter = 12
```

The power method with shift requires in this case a lower number of iterations (1 versus 3) and almost half the cost than the usual power method (also accounting for the extra time needed to compute the LU factorization of A off-line).

Exercise 6.9 It holds

$$\mathrm{A}^{(k)} = \mathrm{Q}^{(k+1)}\mathrm{R}^{(k+1)} \text{ and } \mathrm{A}^{(k+1)} = \mathrm{R}^{(k+1)}\mathrm{Q}^{(k+1)}$$

and then

$$(\mathrm{Q}^{(k+1)})^T\mathrm{A}^{(k)}\mathrm{Q}^{(k+1)} = \mathrm{R}^{(k+1)}\mathrm{Q}^{(k+1)} = \mathrm{A}^{(k+1)}.$$

Since $(\mathrm{Q}^{(k+1)})^T = (\mathrm{Q}^{(k+1)})^{-1}$ we can conclude that matrix $\mathrm{A}^{(k)}$ is similar to $\mathrm{A}^{(k+1)}$ for any $k \geq 0$.

Exercise 6.10 We can use the command `eig` in the following way: `[X,D]=eig (A)`, where `X` is the matrix whose columns are the unit eigenvectors of `A` and `D` is a diagonal matrix whose elements are the eigenvalues of `A`. For the matrices A and B of Exercise 6.7 we should execute the following instructions:

```
A=[2 -1/2 0 -1/2; 0 4 0 2; -1/2 0 6 1/2; 0 0 1 9];
sort(eig(A))
ans =
    2.0000
    4.0268
    5.8003
    9.1728
B=[-5 0 1/2 1/2; 1/2 2 1/2 0; 0 1 0 1/2; 0 1/4 1/2 3];
sort(eig(B))
ans =
   -4.9921
   -0.3038
    2.1666
    3.1292
```

The conclusions drawn on the basis of Proposition 6.1 are quite coarse.

10.7 Chapter 7

Exercise 7.1 By direct inspection on the plot of function f we find that there is a single minimizer in the interval $[-2, 1]$. We use the following instructions to call Program 7.7:

```
a=-2; b=1; tol=1.e-8; kmax=100;
[xmin,fmin,iter]=golden(f,a,b,tol,kmax)
```

Note that the tolerance for the stopping test is set to 10^{-8}. After 42 iterations we obtain `xmin=-3.660253989004456e-01` and `fmin=-1.194742596743503`. The method converges linearly (see (7.19)).
Using now the MATLAB command `fminbnd` with the instructions:

```
options=optimset('TolX',1.e-8);
[xminf,fminf,exitflag,output]=fminbnd(f,a,b,options)
```

the same problem is solved by the golden section method with quadratic interpolation. In this case convergence is achieved in 9 iterations to the point `xmin=-3.660254076197302e-01`.

Exercise 7.2 Given $\gamma_i(t) = (x_i(t), y_i(t))$, for $i = 1, 2$, we need to minimize the distance

$$d(t) = \sqrt{(x_1(t) - x_2(t))^2 + (y_1(t) - y_2(t))^2}$$

or, equivalently, its square as function of t. To solve this one dimensional minimum problem we can use the golden section method with quadratic interpolation implemented in the function `fminbnd`. Using the following instructions

```
x1=@(t)7*cos(t/3+pi/2)+5; y1=@(t)-4*sin(t/3+pi/2)-3;
x2=@(t)6*cos(t/6-pi/3)-4; y2=@(t)-6*sin(t/6-pi/3)+5;
d=@(t)(x1(t)-x2(t))^2+(y1(t)-y2(t))^2;
ta=0; tb=20; options=optimset('TolX',1.e-8);
[tmin,dmin,exitflag,output]=fminbnd(d,ta,tb,options)
```

we converge after 10 iterations to the solution `tmin=8.438731484275010`. At that time, the two ships stand at minimal distance `dmin=5.691754805947144` nautical miles, eigth hours and a half after their departure.

Exercise 7.3 We define the cost function and represent it together with its contour lines on a circular domain centered at (-1,0) with radius 3 by the following instructions:

```
fun=@(x) x(1)^4+x(2)^4+x(1)^3+3*x(1)*...
         x(2)^2-3*x(1)^2-3*x(2)^2+10;
[r,theta]=meshgrid(0:.1:3,0:pi/25:2*pi);
x1=r.*cos(theta)-1; y1=r.*sin(theta);
[n,m]=size(x1);z1=zeros(n,m);
for i=1:n, for j=1:m
    z1(i,j)=fun([x1(i,j);y1(i,j)]);
end, end
figure(1); clf;  p1=mesh(x1,y1,z1);
set(p1,'Edgecolor',[0,1,1]); hold on
contour(x1,y1,z1,100,'Linecolor',[0.8,0.8,0.8]);
```

By a direct inspection we see that the cost function features a local maximizer, a saddle point and two global minimizers (being this function even with respect to the x_2 variable). Choosing $\mathbf{x}^{(0)} = (-3, 0)$ and setting a tolerance $\varepsilon = 10^{-8}$ for the stopping test, using the commands:

```
x0=[-3;0]; options=optimset('TolX',1.e-8);
[xm,fval,exitf,out]=fminsearch(fun,x0,options)
```

we find the minimizer `xm=[-2.1861e+00, 2.1861e+00]` after 181 iterations and having used 353 function evaluations. The second minimizer is therefore `xm=[-2.1861e+00, -2.1861e+00]` because of the parity property of the function.

We warn the reader that choosing `x0=[1;0]`, the `fminsearch` MATLAB function converges to the local maximizer $(.75000, .61237)$ instead than to the minimizer, whereas the `fminsearch` Octave function still converges to the minimizer $(-2.1861, 2.1861)$.

Exercise 7.4 Let us write the sequence $x^{(k+1)} = x^{(k)} + \alpha_k d^{(k)}$ as

$$x^{(k+1)} = x^{(0)} + \sum_{\ell=0}^{k} \alpha_\ell d^{(\ell)}.$$

Since $x^{(0)} = 3/2$ we find

$$\begin{aligned} x^{(k+1)} &= \frac{3}{2} + \left(2 + \frac{2}{3^{k+1}}\right)(-1)^{k+1} = \frac{3}{2} - 2\sum_{\ell=0}^{k}(-1)^\ell - \frac{1}{2} - \frac{1}{6}\left(-\frac{1}{3}\right)^k \\ &= (-1)^{k+1}\left(1 + \frac{1}{6\cdot 3^k}\right). \end{aligned}$$

Note that $x^{(k)}$ does not converge to zero eventhough the sequence $f(x^{(k)})$ is decreasing, as can be seen from Figure 10.8, left. When the points $x^{(k)}$ are near to $+1$ and -1, the first Wolfe condition (7.43) is not fulfilled since the variation of f between two steps becomes infinitesimal while the steplength is about the same (*circa* 2).

Exercise 7.5 By proceeding as done in the previous Exercise, we find $x^{(0)} = -2$ and $x^{(k+1)} = -2 + (1 - 3^{-k})/2 \to -3/2$ when $k \to \infty$. Also in this case

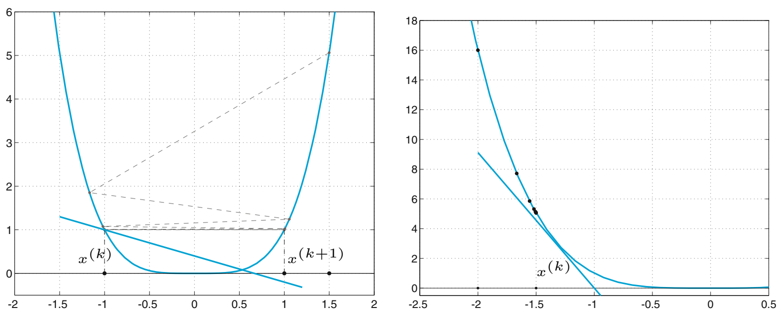


Figure 10.8. At left, the sequence yielded by the descent method of Exercise 7.4. Taking $x^{(k)} \simeq -1$, the point $(x^{(k+1)}, f(x^{(k+1)}))$ should stay beneath the blue straight line in order to satisfy the first Wolfe's condition with $\sigma = 0.2$; on the contrary it lies largely above, indeed $(x^{(k+1)}, f(x^{(k+1)})) \simeq (1, 1)$. At right, the sequence generated for Exercise 7.5. The point $(x^{(k+1)}, f(x^{(k+1)}))$ should stay at the right to the point where the blue straight line is tangent to the blue curve in order for the second Wolfe's condition with $\delta = 0.9$ to be satisifed; instead, it is close to $(-1.5, 5.06)$

the sequence of values $f(x^{(k)})$ is decreasing as we can see in Figure 10.8, right. When the points $x^{(k)}$ are close to $-3/2$, the second Wolfe's condition (7.43) is not satisfied as $f'(x^{(k+1)})$ (with its own sign) should be larger than $\delta f'(x^{(k)})$.

Exercise 7.6 After the following initializations

```
fun=@(x) 100*(x(2)-x(1)^2)^2+(1-x(1))^2;
grad=@(x) [-400*(x(2)-x(1)^2)*x(1)-2*(1-x(1));
            200*(x(2)-x(1)^2)];
hess=@(x) [-400*x(2)+1200*x(1)^2+2, -400*x(1);
           -400*x(1), 200];
x0=[-1.2,1]; tol=1.e-8; kmax=500;
```

we call Program 7.3 using the following instructions:

```
meth=1; % Newton
[x1,err1,k1]=descent(fun,grad,x0,tol,kmax,meth,hess);
meth=2; H0=eye(length(x0)); % BFGS
[x2,err2,k2]=descent(fun,grad,x0,tol,kmax,meth,H0);
meth=3; %gradient
[x3,err3,k3]=descent(fun,grad,x0,tol,kmax,meth);
meth=41; % FR  conjugate gradient
[x41,err41,k41]=descent(fun,grad,x0,tol,kmax,meth);
meth=42; % PR conjugate gradient
[x42,err42,k42]=descent(fun,grad,x0,tol,kmax,meth);
meth=43; % HS conjugate gradient
[x43,err43,k43]=descent(fun,grad,x0,tol,kmax,meth);
```

All the methods converge to the same global minimizer $(1, 1)$, precisely:

```
Newton:  k1  =  22,  err = 1.8652e-12
BFGS:    k2  =  35,  err = 1.7203e-09
Grad:    k3  = 352,  err = 8.1954e-09
CG-FR:   k41 = 284,  err = 5.6524e-10
```

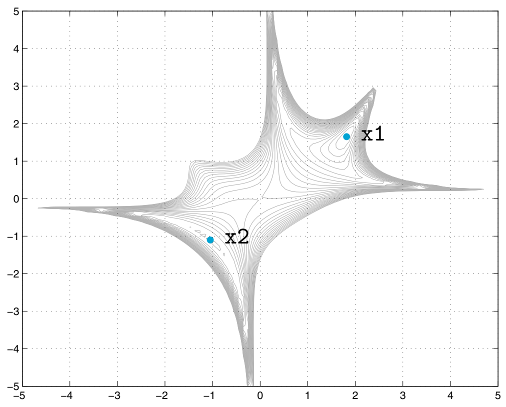


Figure 10.9. Contour lines comprised between the values 0 and 20 of the cost function of Exercise 7.7

```
CG-PR:   k42 = 129,  err = 5.8148e-09
CG-HS:   k43 =  65,  err = 9.8300e-09
```

The number of iterations (`k1, k2, ..., k43`) is in accordance with the theoretical convergence rate of the various methods: quadratic for Newton, superlinear for BFGS, linear for the others. The variable `err` contains the last value of the error estimation used for the stopping test.

Exercise 7.7 By evaluating the function $f(\mathbf{x})$ on the square $[-5, 5]^2$ and graphically representing the contour lines corresponding to the values within the interval $[0, 20]$, we see that it features a saddle point near $(0, 0)$ and two local minimizers, one (`x2`) close to $(-1, -1)$, the other (`x1`) to $(2, 2)$ (see Figure 10.9). (One of them will coincide with the global minimizer we are looking for.) Using `tol=1.e-5` as tolerance for the stopping test and 100 as maximum number of iterations, we take `delta0=0.5` as initial radius for the trust region method implemented in Program 7.4. After having defined the function handle of the cost function and its gradient, we set `meth=2` for both Programs 7.4 and 7.3 in such a way that they use quasi-Newton descent directions (and `hess=eye(2)`). Choosing $\mathbf{x}_0 = (2, -1)$, the trust-region method converges in 28 iterations to the point `x1=(1.8171, 1.6510)`, while the BFGS method converges in 27 iterations to the other local minimizer `x2=(-5.3282e-01, -5.8850e-01)`. Correspondingly, $f($`x1`$) \simeq 3.6661$ and $f($`x2`$) \simeq 8.2226$. Taking instead $\mathbf{x}^{(0)} = (2, 1)$, both methods converge to the global minimizer `x1` in 11 iterations.

Exercise 7.8 Computing the stationary points of $\tilde{f}_k(\mathbf{x}) = \dfrac{1}{2}\|\widetilde{\mathbf{R}}_k(\mathbf{x})\|^2$ amounts to solve the linear system

$$\nabla \tilde{f}_k(\mathbf{x}) = \mathrm{J}_{\widetilde{\mathbf{R}}_k}(\mathbf{x})^T \widetilde{\mathbf{R}}_k(\mathbf{x}) = \mathbf{0}. \tag{10.4}$$

Thanks to the definition (7.64), $\mathrm{J}_{\widetilde{\mathbf{R}}_k}(\mathbf{x}) = \mathrm{J}_{\mathbf{R}}(\mathbf{x}^{(k)})$ for all $\mathbf{x} \in \mathbb{R}^n$ and system (10.4) becomes

$$\mathrm{J}_{\mathbf{R}}(\mathbf{x}^{(k)})^T \mathbf{R}(\mathbf{x}^{(k)}) + \mathrm{J}_{\mathbf{R}}(\mathbf{x}^{(k)})^T \mathrm{J}_{\mathbf{R}}(\mathbf{x}^{(k)})(\mathbf{x} - \mathbf{x}^{(k)}) = \mathbf{0},$$

that is (7.63).

Exercise 7.9 We must show that $\boldsymbol{\delta}\mathbf{x}^{(k)}$ fulfills conditions (7.34). We recall that for every rectangular matrix A having full rank, the square matrix $\mathrm{A}^T\mathrm{A}$ is symmetric and positive definite.

Let us prove $(7.34)_2$. From $\nabla f(\mathbf{x}^{(k)}) = \mathrm{J}_{\mathbf{R}}(\mathbf{x}^{(k)})^T \mathbf{R}(\mathbf{x}^{(k)})$, it follows that $\nabla f(\mathbf{x}^{(k)}) = \mathbf{0}$ iff $\mathbf{R}(\mathbf{x}^{(k)}) = \mathbf{0}$ (as $\mathrm{J}_{\mathbf{R}}(\mathbf{x}^{(k)})$ has full rank) then $\boldsymbol{\delta}\mathbf{x}^{(k)} = \mathbf{0}$ thanks to $(7.63)_1$.

Suppose now that $\mathbf{R}(\mathbf{x}^{(k)}) \neq \mathbf{0}$. Then

$$\begin{aligned}
&(\boldsymbol{\delta}\mathbf{x}^{(k)})^T \nabla f(\mathbf{x}^{(k)}) = \\
&\quad -\left\{ \left[\mathrm{J}_{\mathbf{R}}(\mathbf{x}^{(k)})^T \mathrm{J}_{\mathbf{R}}(\mathbf{x}^{(k)}) \right]^{-1} \mathrm{J}_{\mathbf{R}}(\mathbf{x}^{(k)})^T \mathbf{R}(\mathbf{x}^{(k)}) \right\}^T \mathrm{J}_{\mathbf{R}}(\mathbf{x}^{(k)})^T \mathbf{R}(\mathbf{x}^{(k)}) \\
&\quad -\left(\mathrm{J}_{\mathbf{R}}(\mathbf{x}^{(k)})^T \mathbf{R}(\mathbf{x}^{(k)}) \right)^T \left[\mathrm{J}_{\mathbf{R}}(\mathbf{x}^{(k)})^T \mathrm{J}_{\mathbf{R}}(\mathbf{x}^{(k)}) \right]^{-1} \left(\mathrm{J}_{\mathbf{R}}(\mathbf{x}^{(k)})^T \mathbf{R}(\mathbf{x}^{(k)}) \right) < 0,
\end{aligned}$$

that is $(7.34)_1$ is fulfilled.

Exercise 7.10 Setting $r_i(\mathbf{x}) = x_1 + x_2 t_i + x_3 t_i^2 + x_4 e^{-x_5 t_i} - y_i$, for $i = 1, \ldots, 8$, the desired coefficients $x_1, \ldots, x_5$, are those for which the associated function (7.61) attains its minimum. We call Program 7.5 using the following instructions:

```
t= [0.055;0.181;0.245;0.342;0.419;0.465;0.593;0.752];
y= [2.80;1.76;1.61;1.21;1.25;1.13;0.52;0.28];
tol=1.e-12; kmax=500;
x0=[2,-2.5,-.2,5,35];
[x,err,iter]= gaussnewton(@mqnlr,@mqnljr,...
    x0,tol,kmax,t,y);
```

where `mqnlr` and `mqnljr` are the functions which define $\mathbf{R}(\mathbf{x})$ and $\mathrm{J}_{\mathbf{R}}(\mathbf{x})$ respectively:

```
function r=mqnlr(x,t,y)
m=length(t); n=length(x);
r=zeros(m,1);
for i=1:m
r(i)=sqrt(2)*(x(1)+t(i)*x(2)+t(i)^2*x(3)+...
    x(4)*exp(-t(i)*x(5))-y(i));
end

function jr=mqnljr(x,t,y)
m=length(t); n=length(x); jr=zeros(m,n);
for i=1:m
jr(i,1)=1; jr(i,2)=t(i);
jr(i,3)=t(i)^2; jr(i,4)=exp(-t(i)*x(5));
jr(i,5)=-t(i)*x(4)*exp(-t(i)*x(5));
end
jr=jr*sqrt(2);
```

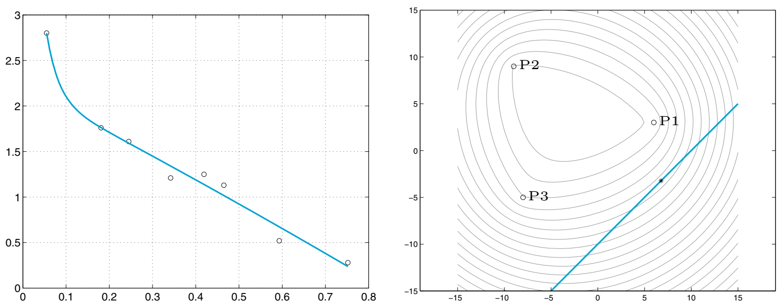


Figure 10.10. Left: the data and the solution of the Exercise 7.10. Right: the solution of the Exercise 7.12. The grey curves represent the contour lines of the cost function. The admissibility domain Ω is the unbounded portion of the plane beneath the blue straight line

After 19 iterations convergence is achieved to the point
`x=[2.2058e+00 -2.4583e+00 -2.1182e-01 5.2106e+00 3.5733e+01]`.
At that point the residual is far from zero, actually $f(x) = 1.8428e-01$. Nevertheless we can classify the given problem as a small residual problem, therefore convergence is linear. For the same problem, Newton's method (7.31) converges in 8 iterations. If the initial point is not close enough to the minimizer, e.g. if `x0 = [1,1,1,1,10]`, the Gauss-Newton method fails to converge, while the damped Gauss-Newton method converges in 21 iterations. In Figure 10.10, left, we plot function $\phi(t)$ whose coefficients $x_1, \ldots, x_5$ are those computed numerically. The empty circles represent the distribution of data (t_i, y_i).

Exercise 7.11 Starting from $\Phi(\mathbf{x}) = \frac{1}{2}\|\mathbf{R}(\mathbf{x})\|^2$ a quadratic approximation of Φ around $\mathbf{x}^{(k)}$ reads

$$\tilde{\Phi}_k(\mathbf{s}) = \Phi(\mathbf{x}^{(k)}) + \mathbf{s}^T\nabla\Phi(\mathbf{x}^{(k)}) + \frac{1}{2}\mathbf{s}^T\mathrm{H}_k\mathbf{s} \qquad \forall\mathbf{s}\in\mathbb{R}^n,$$

where H_k is a suitable approximation of the Hessian of Φ. By exploiting (7.62) and taking

$$\mathrm{H}_k = \mathrm{J}_{\mathbf{R}}(\mathbf{x}^{(k)})^T\mathrm{J}_{\mathbf{R}}(\mathbf{x}^{(k)}),$$

it holds

$$\begin{aligned}\tilde{\Phi}_k(\mathbf{s}) &= \frac{1}{2}\|\mathbf{R}(\mathbf{x}^{(k)})\|^2 + \mathbf{s}^T\mathrm{J}_{\mathbf{R}}(\mathbf{x}^{(k)})^T\mathbf{R}(\mathbf{x}^{(k)}) + \frac{1}{2}\mathbf{s}^T\mathrm{J}_{\mathbf{R}}(\mathbf{x}^{(k)})^T\mathrm{J}_{\mathbf{R}}(\mathbf{x}^{(k)})\mathbf{s}\\ &= \frac{1}{2}\|\mathbf{R}(\mathbf{x}^{(k)}) + \mathrm{J}_{\mathbf{R}}(\mathbf{x}^{(k)})\mathbf{s}\|^2\\ &= \frac{1}{2}\|\tilde{\mathbf{R}}_k(\mathbf{x})\|^2.\end{aligned}$$

In conclusion, $\tilde{\Phi}_k$ can be regarded as a quadratic model of Φ around $\mathbf{x}^{(k)}$, obtained by replacing $\mathbf{R}(\mathbf{x})$ with $\tilde{\mathbf{R}}_k(\mathbf{x})$.

Exercise 7.12 We need to solve the minimization problem (7.2) with cost function $f(x,y) = \sum_{i=1}^{3} v_i \sqrt{(x-x_i)^2 + (y-y_i)^2}$ and admissibility domain $\Omega = \{(x,y) \in \mathbb{R}^2 : \ y \leq x - 10\}$. The values v_i represent the number of journeys toward the selling point P_i.
We define first the cost function and the constraint functions, then we call Program `penalty.m`, using the following instructions:

```
x1=[6; 3]; x2=[-9;9]; x3=[-8;-5]; v=[140;134;88];
d=@(x)v(1)*sqrt((x(1)-x1(1)).^2+(x(2)-x1(2)).^2)+...
     v(2)*sqrt((x(1)-x2(1)).^2+(x(2)-x2(2)).^2)+...
     v(3)*sqrt((x(1)-x3(1)).^2+(x(2)-x3(2)).^2);
g=@(x)[x(1)-x(2)-10];
meth=0; x0=[10;-10]; tol=1.e-8; kmax=200;kmaxd=200;
[xmin,err,k]=penalty(d,[],[],[],g,[],x0,tol,...
                  kmax,kmaxd,meth);
```

This program makes use of the penalty algorithm coupled with the Nelder and Mead method for unconstrained minimization. We have not used descent method since the cost function features non-differentiable points, moreover the matrices H_k used for the direction $\mathbf{d}^{(k)}$ may be ill-conditioned. The optimal location where to place the warehouse has coordinates `xmin=[6.7734,-3.2266]`. Convergence is achieved after 13 iterations of the penalty method.

Exercise 7.13 Since no inequality constraint is present, problem can be rewritten under the form (7.77) and then we can proceed as done in Example 7.14. Matrix C has rank 2 and its kernel $ker(\mathrm{C}) = \{\mathbf{z} = \alpha[1,1,1]^T \alpha \in \mathbb{R}\}$ has dimension 1. Matrix A is symmetric; as $\sum_{i,j=1}^{3} a_{ij} > 0$, it is positive definite when restricted to the kernel of C. We built matrix $\mathrm{M} = [\mathrm{A}, \ -\mathrm{C}^T; \mathrm{C}, \ 0]$ and the right hand side $\mathbf{f} = [-\mathbf{b}, \mathbf{d}]^T$, then we solve the linear system (7.77) using the instructions:

```
A=[2,-1,1;-1,3,4;1,4,1]; b=[1;-2;-1];
C=[2,-2,0;2,1,-3]; d=[1;1];
M=[A -C'; C, zeros(2)]; f=[-b;d];
xl=M\f;
```

We obtain the solution

```
xl =
   5.7143e-01
   7.1429e-02
   7.1429e-02
   1.0476e+00
   2.3810e-02
```

The first 3 components of `xl` provide the approximation of the minimizer, whereas the Lagrangian multipliers associated to the constraints are given by the last components. The minimum value attained by the cost function is `6.9388e-01`.

Exercise 7.14 We represent the function $v(x,y)$ on the square $[-2.5, 2.5]^2$ and its restriction to the curve $h(x,y) = x^2/4 + y^2 - 1 = 0$ representing the constraint in Figure 10.11. As we can see, several local maximizers exist, the global one lying in a neighborhoud of the point (2,0.5).
We use the following instructions to call Program 7.7

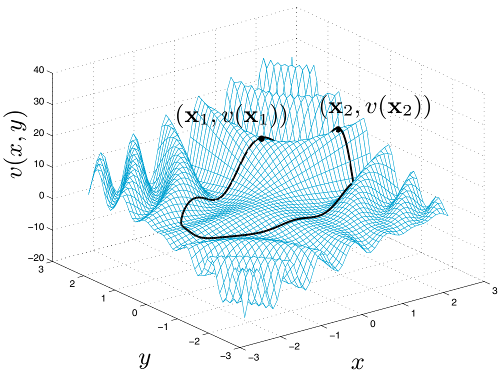


Figure 10.11. The function $v(x, y)$ of Exercise 7.14 and the two maxima computed using the augmented Lagrangian method

```
fun=@(x)-(sin(pi*x(1)*x(2))+1)*(2*x(1)+3*x(2)+4);
grad_fun=@(x)[-pi*x(2)*cos(pi*x(1)*x(2))*...
    (2*x(1)+3*x(2)+4)-(sin(pi*x(1)*x(2))+1)*2;
    -pi*x(1)*cos(pi*x(1)*x(2))*(2*x(1)+3*x(2)+4)-...
    (sin(pi*x(1)*x(2))+1)*3];
h=@(x)x(1)^2/4+x(2)^2-1;  grad_h=@(x)[x(1)/2;2*x(2)];
x0=[1;0];  lambda0=1;  tol=1.e-8;  kmax=100;  kmaxd=100;
meth=2;hess=eye(2);
[x,err,k]=auglagrange(fun,grad_fun,h,grad_h,...
     x0,lambda0,tol,kmax,kmaxd,meth,hess)
```

To solve the unconstrained minimization problem for the function $f(x, y) = -v(x, y)$ inside the augmented Lagrangian method, we use BFGS method. Choosing $\mathbf{x}^{(0)} = (1, 0)$, convergence is achieved in 6 iterations to the point $\mathbf{x}_1 = (0.56833, 0.95877)$. The latter is a maximizer but not the global one, as Figure 10.11 shows. Choosing instead $\mathbf{x}^{(0)} = (2, 1)$ we obtain convergence (in 5 iterations) to the point $\mathbf{x}_2 = (1.9242, 0.27265)$; note that $v(\mathbf{x}_1) = 15.94$ while $v(\mathbf{x}_2) = 17.307$, $\mathbf{x}_2$ is therefore the global maximizer.

10.8 Chapter 8

Exercise 8.1 Let us approximate the exact solution $y(t) = \frac{1}{2}[e^t - \sin(t) - \cos(t)]$ of the Cauchy problem (8.85) by the forward Euler method using different values of h: $1/2$, $1/4$, $1/8, \ldots, 1/512$. The associated error is computed by the following instructions:

```
t0=0; y0=0; T=1; f=@(t,y) sin(t)+y;
y=@(t) 0.5*(exp(t)-sin(t)-cos(t));
Nh=2;
for k=1:10;
[tt,u]=feuler(f,[t0,T],y0,Nh);
e(k)=max(abs(u-y(tt)));Nh=2*Nh;
```

```
end
```

Now we apply formula (1.12) to estimate the order of convergence:

```
p=log(abs(e(1:end-1)./e(2:end)))/log(2); p(1:2:end)
p =
    0.7696    0.9273    0.9806    0.9951    0.9988
```

As expected the order of convergence is one. With the same instructions (substituting the call to Program 8.1 with that to Program 8.2) we obtain an estimate of the convergence order of the backward Euler method:

```
p=log(abs(e(1:end-1)./e(2:end)))/log(2); p(1:2:end)
p =
    1.5199    1.0881    1.0204    1.0050    1.0012
```

Exercise 8.2 The numerical solution of the given Cauchy problem by the forward Euler method can be obtained as follows:

```
t0=0; T=1; N=100; f=@(t,y) -t*exp(-y);
y0=0;[t,u]=feuler(f,[t0,T],y0,N);
```

To compute the number of exact significant digits we can estimate the constants L and M which appear in (8.13). Note that, since $f(t, y(t)) < 0$ in the given interval, $y(t)$ is a monotonically decreasing function, vanishing at $t = 0$. Since f is continuous together with its first derivative, we can approximate L as $L = \max_{0\leq t\leq 1} |L(t)|$ with $L(t) = \partial f/\partial y = te^{-y}$. Note that $L(0) = 0$ and $L'(t) > 0$ for all $t \in (0, 1]$. Thus, by using the assumption $-1 < y < 0$, we can take $L = e$.

Similarly, in order to compute $M = \max_{0\leq t\leq 1} |y''(t)|$ with $y'' = -e^{-y} - t^2e^{-2y}$, we can observe that this function has its maximum at $t = 1$, and then $M = e + e^2$. We can draw these conclusions by analyzing the graph of the vector field $\mathbf{v}(t, y) = [v_1, v_2]^T = [1, f(t, y(t))]^T$ associated to the given Cauchy problem. Indeed, the solutions of the differential equation $y'(t) = f(t, y(t))$ are tangential to the vector field $\mathbf{v}$. By the following instructions:

```
[T,Y]=meshgrid(0:0.05:1,-1:0.05:0);
V1=ones(size(T)); V2=-T.*exp(Y); quiver(T,Y,V1,V2)
```

we see that the solution of the Cauchy problem has a nonpositive second derivative whose absolute value grows up with t. This fact leads us to conclude that $M = \max_{0\leq t\leq 1} |y''(t)|$ is reached at $t = 1$.
The same conclusions can be drawn by noticing that the function $-y$ is positive and increasing, since $y \in [-1, 0]$ and $f(t, y) = y' < 0$. Thus, also the functions e^{-y} and t^2e^{-2y} are positive and increasing, while the function $y'' = -e^{-y} - t^2e^{-2y}$ is negative and decreasing. It follows that $M = \max_{0\leq t\leq 1} |y''(t)|$ is obtained at $t = 1$.

From (8.13), for $h = 0.01$ we deduce

$$|u_{100} - y(1)| \leq \frac{e^L - 1}{L}\frac{M}{200} \simeq 0.26.$$

Therefore, there is no guarantee that more than one significant digit be exact. Indeed, we find `u(end)=-0.6785`, while the exact solution ($y(t) = \log(1-t^2/2)$) at $t = 1$ is $y(1) = -0.6931$.

Exercise 8.3 The iteration function is $\phi(u) = u - ht_{n+1}e^{-u}$ and the fixed-point iteration converges if $|\phi'(u)| < 1$. This property is ensured if $h(t_0 + (n+1)h) < e^u$. If we substitute u with the exact solution, we can provide an *a priori* estimate of the value of h. The most restrictive situation occurs when $u = -1$ (see Solution 8.2). In this case the solution of the inequality $(n+1)h^2 < e^{-1}$ is $h < \sqrt{e^{-1}/(n+1)}$.

Exercise 8.4 We repeat the same set of instructions of Solution 8.1, however now we use the program `cranknic` (Program 8.3) instead of `feuler`. According to the theory, we obtain the following result that shows second-order convergence:

```
p=log(abs(e(1:end-1)./e(2:end)))/log(2); p(1:2:end)
p =
     2.0379     2.0023     2.0001     2.0000     2.0000
```

Exercise 8.5 Consider the integral formulation of the Cauchy problem (8.5) in the interval $[t_n, t_{n+1}]$:

$$\begin{aligned} y(t_{n+1}) - y(t_n) &= \int_{t_n}^{t_{n+1}} f(\tau, y(\tau))d\tau \\ &\simeq \frac{h}{2}\left[f(t_n, y(t_n)) + f(t_{n+1}, y(t_{n+1}))\right], \end{aligned}$$

where we have approximated the integral by the trapezoidal formula (4.19). By setting $u_0 = y(t_0)$ and defining u_{n+1} as

$$u_{n+1} = u_n + \frac{h}{2}\left[f(t_n, u_n) + f(t_{n+1}, u_{n+1})\right], \qquad \forall n \geq 0,$$

we obtain precisely the Crank-Nicolson method.

Exercise 8.6 We know that the absolute stability region for the forward Euler method is the circle centered at $(-1, 0)$ with radius equal to 1, that is the set $A = \{z = h\lambda \in \mathbb{C} : \ |1 + h\lambda| < 1\}$. By replacing $\lambda = -1 + i$ we obtain the bound on h: $h^2 - h < 0$, i.e. $h \in (0, 1)$.

Exercise 8.7 Let us rewrite the Heun method in the following (Runge-Kutta like) form:

$$\begin{aligned} &u_{n+1} = u_n + \frac{h}{2}(K_1 + K_2), \\ &K_1 = f(t_n, u_n), \quad K_2 = f(t_{n+1}, u_n + hK_1). \end{aligned} \tag{10.5}$$

We have $h\tau_{n+1}(h) = y(t_{n+1}) - y(t_n) - h(\widehat{K}_1 + \widehat{K}_2)/2$, with $\widehat{K}_1 = f(t_n, y(t_n))$ and $\widehat{K}_2 = f(t_{n+1}, y(t_n) + h\widehat{K}_1)$. Since f is continuous with respect to both arguments, it holds

$$\lim_{h \to 0} \tau_{n+1} = y'(t_n) - \frac{1}{2}[f(t_n, y(t_n)) + f(t_n, y(t_n))] = 0.$$

Therefore, the Heun method is consistent. We prove now that τ_{n+1} is an infinitesimal of second order with respect to h. Suppose that $y \in C^3([t_0, T[)$. For simplicity of notations, we set $y_n = y(t_n)$ for any $n \geq 0$. We have

$$\begin{aligned}\tau_{n+1} &= \frac{y_{n+1} - y_n}{h} - \frac{1}{2}\left[f(t_n, y_n) + f(t_{n+1}, y_n + hf(t_n, y_n))\right] \\ &= \frac{y_{n+1} - y_n}{h} - \frac{1}{2}y'(t_n) - \frac{1}{2}f(t_{n+1}, y_n + hy'(t_n)).\end{aligned}$$

Thanks to the error formula (4.20) related to the trapezoidal rule there exists $\xi_n \in]t_n, t_{n+1}[$ such that

$$y_{n+1} - y_n = \int_{t_n}^{t_{n+1}} y'(t)dt = \frac{h}{2}\left[y'(t_n) + y'(t_{n+1})\right] - \frac{h^3}{12}y'''(\xi_n),$$

therefore

$$\begin{aligned}\tau_{n+1} &= \frac{1}{2}\left(y'(t_{n+1}) - f(t_{n+1}, y_n + hy'(t_n)) - \frac{h^2}{6}y'''(\xi_n)\right) \\ &= \frac{1}{2}\left(f(t_{n+1}, y_{n+1}) - f(t_{n+1}, y_n + hy'(t_n)) - \frac{h^2}{6}y'''(\xi_n)\right).\end{aligned}$$

Moreover, as the function f is Lipschitz continuous with respect to the second variable (see Proposition 8.1), it holds

$$|\tau_{n+1}| \leq \frac{L}{2}|y_{n+1} - y_n - hy'(t_n)| + \frac{h^2}{12}|y'''(\xi_n)|.$$

Finally, by applying the Taylor formula

$$y_{n+1} = y_n + hy'(t_n) + \frac{h^2}{2}y''(\eta_n), \qquad \eta_n \in]t_n, t_{n+1}[,$$

we obtain

$$|\tau_{n+1}| \leq \frac{L}{4}h^2|y''(\eta_n)| + \frac{h^2}{12}|y'''(\xi_n)| \leq Ch^2.$$

The Heun method is implemented in Program 10.2. Using this program, we can verify the order of convergence as in Solution 8.1. Precisely, by the following instructions, we find that the Heun method is second-order accurate with respect to h

```
p=log(abs(e(1:end-1)./e(2:end)))/log(2); p(1:2:end)
ans =
     1.7642     1.9398     1.9851     1.9963     1.9991
```

Program 10.2. rk2: Heun (or RK2) method

```
function [tt,u]=rk2(odefun,tspan,y0,Nh,varargin)
tt=linspace(tspan(1),tspan(2),Nh+1);
h=(tspan(2)-tspan(1))/Nh;  hh=h*0.5;
u=y0;
for t=tt(1:end-1)
  y = u(end,:);
  k1=odefun(t,y,varargin{:});
```

```
   t1 = t + h; y = y + h*k1;
   k2=odefun(t1,y,varargin{:});
   u = [u; u(end,:) + hh*(k1+k2)];
end
tt=tt';
```

Exercise 8.8 Applying the method (10.5) to the model problem (8.28) we obtain $K_1 = \lambda u_n$ and $K_2 = \lambda u_n(1+h\lambda)$. Therefore $u_{n+1} = u_n[1+h\lambda+(h\lambda)^2/2] = u_n p_2(h\lambda)$. To ensure absolute stability we must require that $|p_2(h\lambda)| < 1$, which is equivalent to $0 < p_2(h\lambda) < 1$, since $p_2(h\lambda)$ is positive. Solving the latter inequality, we obtain $-2 < h\lambda < 0$, that is, $h < 2/|\lambda|$, since λ is a real negative number.

Exercise 8.9 We prove the property (8.34), that we call for simplicity $\mathcal{P}_n$, by induction on n. To this aim, it is sufficient to prove that if $\mathcal{P}_1$ holds and if $\mathcal{P}_{n-1}$ implies $\mathcal{P}_n$ for any $n \geq 2$, then $\mathcal{P}_n$ holds for any $n \geq 2$.
It is easily verified that $u_1 = u_0 + h(\lambda_0 u_0 + r_0)$. In order to prove that $\mathcal{P}_{n-1} \Rightarrow \mathcal{P}_n$, it is sufficient to note that $u_n = u_{n-1}(1 + h\lambda_{n-1}) + hr_{n-1}$.

Exercise 8.10 Since $|1 + h\lambda| < 1$, from (8.38) it follows

$$|z_n - u_n| \leq |\rho| \left(\left|1 + \frac{1}{\lambda}\right| + \left|\frac{1}{\lambda}\right| \right).$$

If $\lambda \leq -1$, we have $1/\lambda < 0$ and $1 + 1/\lambda \geq 0$, then

$$\left|1 + \frac{1}{\lambda}\right| + \left|\frac{1}{\lambda}\right| = 1 + \frac{1}{\lambda} - \frac{1}{\lambda} = 1 = \varphi(\lambda).$$

On the other hand, if $-1 < \lambda < 0$, we have $1/\lambda < 1 + 1/\lambda < 0$, then

$$\left|1 + \frac{1}{\lambda}\right| + \left|\frac{1}{\lambda}\right| = -1 - \frac{2}{\lambda} = \left|1 + \frac{2}{\lambda}\right| = \varphi(\lambda).$$

Exercise 8.11 From (8.36) we have

$$|z_n - u_n| \leq \overline{\rho}[a(h)]^n + h\overline{\rho}\sum_{k=0}^{n-1}[a(h)]^{n-k-1}.$$

The result follows using (8.37).

Exercise 8.12 We have

$$h\tau_{n+1}(h) = y(t_{n+1}) - y(t_n) - \frac{h}{6}(\widehat{K}_1 + 4\widehat{K}_2 + \widehat{K}_3),$$
$$\widehat{K}_1 = f(t_n, y(t_n)), \quad \widehat{K}_2 = f(t_n + \tfrac{h}{2}, y(t_n) + \tfrac{h}{2}\widehat{K}_1),$$
$$\widehat{K}_3 = f(t_{n+1}, y(t_n) + h(2\widehat{K}_2 - \widehat{K}_1)).$$

Since f is continuous with respect to both arguments, we obtain

$$\lim_{h\to 0}\tau_{n+1} = y'(t_n) - \frac{1}{6}[f(t_n, y(t_n)) + 4f(t_n, y(t_n)) + f(t_n, y(t_n))] = 0,$$

which proves that the method is consistent.

This method is an explicit Runge-Kutta method of order 3 and is implemented in Program 10.3. As in Solution 8.7, we can derive an estimate of its order of convergence by the following instructions:

```
p=log(abs(e(1:end-1)./e(2:end)))/log(2); p(1:2:end)
ans =
    2.7306     2.9330     2.9833     2.9958     2.9990
```

Program 10.3. rk3: explicit Runge-Kutta method of order 3

```
function [tt,u]=rk3(odefun,tspan,y0,Nh,varargin);
tt=linspace(tspan(1),tspan(2),Nh+1);
h=(tspan(2)-tspan(1))/Nh; hh=h*0.5; h2=2*h;
u=y0; h6=h/6;
for t=tt(1:end-1)
  y = u(end,:);
  k1=odefun(t,y,varargin{:});
  t1 = t + hh; y1 = y + hh* k1;
  k2=odefun(t1,y1,varargin{:});
  t1 = t + h; y1 = y + h*(2*k2-k1);
  k3=odefun(t1,y1,varargin{:});
  u = [u; u(end,:) + h6*(k1+4*k2+k3)];
end
tt=tt';
```

Exercise 8.13 By following the same arguments used in Solution 8.8, we obtain the relation

$$u_{n+1} = u_n[1 + h\lambda + \frac{1}{2}(h\lambda)^2 + \frac{1}{6}(h\lambda)^3] = u_n p_3(h\lambda).$$

By inspection of the graph of p_3, obtained with the instruction

```
c=[1/6 1/2 1 1]; z=[-3:0.01:1];
p=polyval(c,z); plot(z,abs(p))
```

we deduce that $|p_3(h\lambda)| < 1$, provided that $-2.5 < h\lambda < 0$.

Exercise 8.14 The method (8.87) applied to the model problem (8.28) with $\lambda \in \mathbb{R}^-$ gives the equation $u_{n+1} = u_n(1+h\lambda+(h\lambda)^2)$. By solving the inequality $|1 + h\lambda + (h\lambda)^2| < 1$ we find $-1 < h\lambda < 0$.

Exercise 8.15 To solve Problem 8.1 with the given values, we repeat the following instructions with `N=10` and `N=20`:

```
f=@(t,y) -1.68e-9*y^4+2.6880;
[tc,uc]=cranknic(f,[0,200],180,N);
[tp,up]=rk2(f,[0,200],180,N);
```

The graphs of the computed solutions are shown in Figure 10.12.

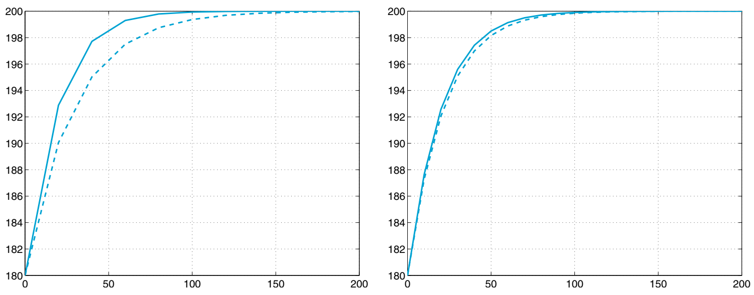


Figure 10.12. Computed solutions with `N = 10` (*left*) and `N = 20` (*right*) for the Cauchy problem of Solution 8.15: the solutions computed by the Crank-Nicolson method (*solid line*), and by the Heun method (*dashed line*)

Exercise 8.16 Heun method applied to the model problem (8.28), gives

$$u_{n+1} = u_n \left(1 + h\lambda + \frac{1}{2}h^2\lambda^2\right).$$

In the complex plane the boundary of the region of absolute stability is the set of points $h\lambda = x+iy$ such that $|1+h\lambda+h^2\lambda^2/2|^2 = 1$. This equation is satisfied by the pairs (x, y) such that $f(x, y) = x^4+y^4+2x^2y^2+4x^3+4xy^2+8x^2+8x = 0$. We can represent this curve as the 0-contour line of the function $z = f(x, y)$. This can be done by means of the following instructions:

```
f=@(x,y)[x.^4+y.^4+2*(x.^2).*(y.^2)+...
         4*x.*y.^2+4*x.^3+8*x.^2+8*x];
[x,y]=meshgrid([-2.1:0.1:0.1],[-2:0.1:2]);
contour(x,y,f(x,y),[0 0]); grid on
```

The command `meshgrid` draws in the rectangle $[-2.1, 0.1] \times [-2, 2]$ a grid with 23 equispaced nodes in the x-direction, and 41 equispaced nodes in the y-direction. With the command `contour` we plot the contour line of $f(x, y)$ corresponding to the value $z = 0$ (made precise in the input vector `[0 0]` of `contour`). In Figure 10.13 the solid line delimitates the region of absolute stability of the Heun method. This region is larger than the absolute stability region of the forward Euler method (which corresponds to the interior of the dashed circle). Both curves are tangent to the imaginary axis at the origin $(0, 0)$.

`contour`

Exercise 8.17 We use the following instructions:

```
t0=0; y0=0; f=@(t,y)cos(2*y);
y=@(t) 0.5*asin((exp(4*t)-1)./(exp(4*t)+1));
T=1; N=2; for k=1:10;
[tt,u]=rk2(f,[t0,T],y0,N);
e(k)=max(abs(u-y(tt))); N=2*N; end
p=log(abs(e(1:end-1)./e(2:end)))/log(2); p(1:2:end)

     2.4733     2.1223     2.0298     2.0074     2.0018
```

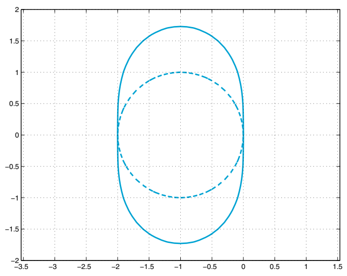


Figure 10.13. Boundaries of the regions of absolute stability for the Heun method (*solid line*) and the forward Euler method (*dashed line*). The corresponding regions lie at the interior of the boundaries

As expected, we find that the order of convergence of the method is 2. However, the computational cost is comparable with that of the forward Euler method, which is first-order accurate only.

Exercise 8.18 The second-order differential equation of this exercise is equivalent to the following first-order system:

$$x'(t) = z(t), \qquad z'(t) = -5z(t) - 6x(t),$$

with $x(0) = 1$, $z(0) = 0$. We use the Heun method as follows:

```
t0=0; y0=[1 0]; T=5;
[t,u]=rk2(@fspring,[t0,T],y0,N);
```

where `N` is the number of nodes and `fspring.m` is the following function:

```
function fn=fspring(t,y)
b=5;
k=6;
[n,m]=size(y);
fn=zeros(n,m);
fn(1)=y(2);
fn(2)=-b*y(2)-k*y(1);
```

In Figure 10.14 we show the graphs of the two components of the solution, computed with `N=20` and `N=40` and compare them with the graph of the exact solution $x(t) = 3e^{-2t} - 2e^{-3t}$ and that of its first derivative.

Exercise 8.19 The second-order system of differential equations is reduced to the following first-order system:

$$\begin{cases} x'(t) = z(t), \\ y'(t) = v(t), \\ z'(t) = 2\omega \sin(\Psi) v(t) - k^2 x(t), \\ v'(t) = -2\omega \sin(\Psi) z(t) - k^2 y(t). \end{cases} \tag{10.6}$$

If we suppose that the pendulum at the initial time $t_0 = 0$ is at rest in the position $(1, 0)$, the system (10.6) must be given the following initial conditions:

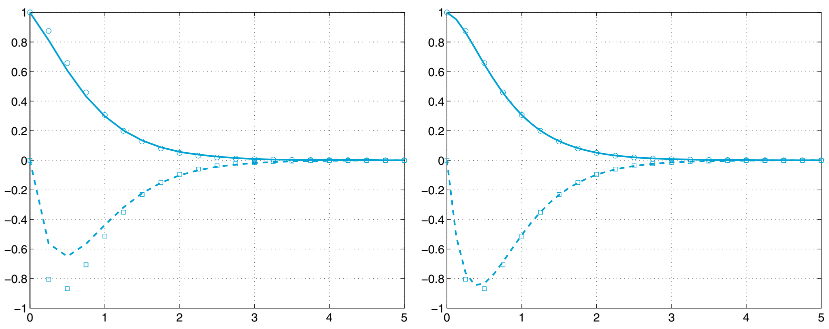


Figure 10.14. Approximations of $x(t)$ (*solid line*) and $x'(t)$ (*dashed line*) computed with `N=20` (*at left*) and `N=40` (*at right*). Small circles and squares refer to the exact functions $x(t)$ and $x'(t)$, respectively

$$x(0) = 1,\ y(0) = 0,\ z(0) = 0,\ v(0) = 0.$$

Setting $\Psi = \pi/4$, which is the average latitude of the Northern Italy, we use the forward Euler method as follows:

```
[t,u]=feuler(@ffoucault,[0,300],[1 0 0 0],N);
```

where `N` is the number of steps and `ffoucault.m` is the following function:

```
function fn=ffoucault(t,y)
l=20; k2=9.8/l; psi=pi/4; omega=7.29*1.e-05;
[n,m]=size(y); fn=zeros(n,m);
fn(1)=y(3);  fn(2)=y(4);
fn(3)=2*omega*sin(psi)*y(4)-k2*y(1);
fn(4)=-2*omega*sin(psi)*y(3)-k2*y(2);
```

By some numerical experiments we conclude that the forward Euler method cannot produce acceptable solutions for this problem even for very small h. For instance, on the left of Figure 10.15 we show the graph, in the phase plane (x, y), of the motion of the pendulum computed with `N=30000`, that is, $h = 1/100$. As expected, the rotation plane changes with time, but also the amplitude of the oscillations increases. Similar results can be obtained for smaller h and using the Heun method. In fact, the model problem corresponding to the problem at hand has a coefficient λ that is purely imaginary. The corresponding solution (a sine function) is bounded for any t, however it doesn't tend to zero.

Unfortunately, both the forward Euler and Heun methods feature a region of absolute stability that doesn't include any point of the imaginary axis (with the exception of the origin). Thus, to ensure the absolute stability one should choose the prohibited value $h = 0$.

To get an acceptable solution we should use a method whose region of absolute stability includes a portion of the imaginary axis. This is the case, for instance, for the adaptive Runge-Kutta method of order 3, implemented in the MATLAB function `ode23`. We can invoke it by the following command:

```
[t,u]=ode23(@ffoucault,[0,300],[1 0 0 0]);
```

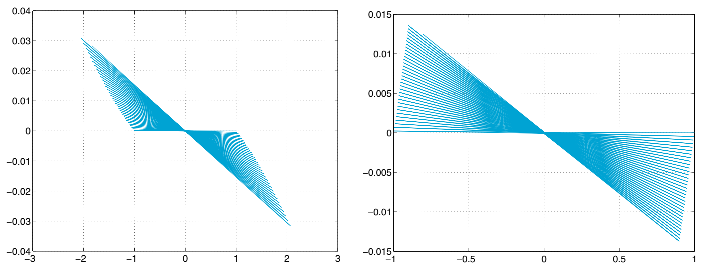


Figure 10.15. Trajectories on the phase plane for the Foucault pendulum of Solution 8.19 computed by the forward Euler method (*left*) and the third-order adaptive Runge-Kutta method (*right*)

In Figure 10.15 (*right*) we show the solution obtained using only 1022 integration steps. Note that the numerical solution is in good agreement with the exact one.

Exercise 8.20 We fix the right hand side of the problem in the following *function*

```
function fn=baseball(t,y)
phi = pi/180;   omega = 1800*1.047198e-01;
B = 4.1*1.e-4; g = 9.8;
[n,m]=size(y); fn=zeros(n,m);
vmodule = sqrt(y(4)^2+y(5)^2+y(6)^2);
Fv = 0.0039+0.0058/(1+exp((vmodule-35)/5));
fn(1)=y(4);
fn(2)=y(5);
fn(3)=y(6);
fn(4)=-Fv*vmodule*y(4)+...
       B*omega*(y(6)*sin(phi)-y(5)*cos(phi));
fn(5)=-Fv*vmodule*y(5)+B*omega*y(4)*cos(phi);
fn(6)=-g-Fv*vmodule*y(6)-B*omega*y(4)*sin(phi);
```

At this point we only need to recall `ode23` as follows:

```
[t,u]=ode23(@baseball,[0 0.4],...
      [0 0 0 38*cos(pi/180) 0 38*sin(pi/180)]);
```

Using command `find` we approximately compute the time at which the altitude becomes negative, which corresponds to the exact time of impact with the ground:

```
n=max(find(u(:,3)>=0)); t(n)
ans =
    0.1066
```

In Figure 10.16 we report the trajectories of the baseball with an inclination of 1 and 3 degrees represented on the plane x_1x_3 and on the $x_1x_2x_3$ space, respectively.

Exercise 8.21 Let us define the function

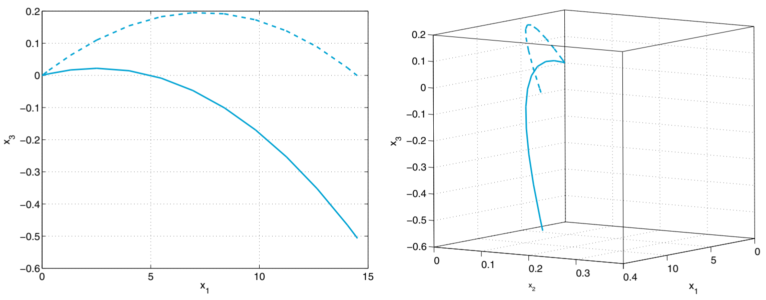


Figure 10.16. The trajectories followed by a baseball launched with an initial angle of 1 degree (*solid line*) and 3 degrees (*dashed line*), respectively

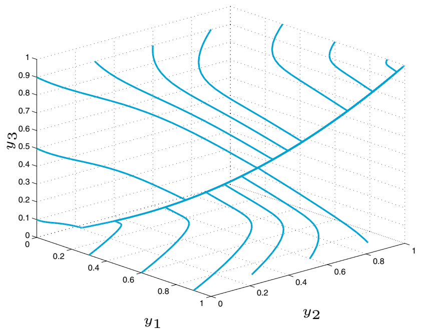


Figure 10.17. Trajectories of the model (8.88) corresponding to several intial data and with $\varepsilon = 10^{-2}$

```
function f=fchem3(t,y)
e=1.e-2;
[n,m]=size(y);f=zeros(n,m);
f(1)=1/e*(-5*y(1)-y(1)*y(2)+5*y(2)^2+...
     y(3))+y(2)*y(3)-y(1);
f(2)=1/e*(10*y(1)-y(1)*y(2)-10*y(2)^2+y(3))...
     -y(2)*y(3)+y(1);
f(3)=1/e*(y(1)*y(2)-y(3))-y(2)*y(3)+y(1);
```

and execute the following instructions

```
y0=[1,0.5,0]; tspan=[0,10];
[t1,y1]=ode23(@fchem3,tspan,y0);
[t2,y2]=ode23s(@fchem3,tspan,y0);
fprintf('Passi ode23=%d, passi ode23s=%d\n',...
length(t1),length(t2))
```

`ode23` requires 8999 steps while `ode23s` only 43. Consequently we can state that the given problem is stiff. The computed numerical solutions are shown in Figure 10.17.

10.9 Chapter 9

Exercise 9.1 We can verify directly that $\mathbf{x}^T \mathrm{A}\mathbf{x} > 0$ for all $\mathbf{x} \neq \mathbf{0}$. Indeed,

$$[x_1\ x_2\ \ldots\ x_{N-1}\ x_N] \begin{bmatrix} 2 & -1 & 0 & \ldots & 0 \\ -1 & 2 & \ddots & & \vdots \\ 0 & \ddots & \ddots & -1 & 0 \\ \vdots & & -1 & 2 & -1 \\ 0 & \ldots & 0 & -1 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{N-1} \\ x_N \end{bmatrix}$$
$$= 2x_1^2 - 2x_1x_2 + 2x_2^2 - 2x_2x_3 + \ldots - 2x_{N-1}x_N + 2x_N^2.$$

The last expression is equivalent to $(x_1-x_2)^2+\ldots+(x_{N-1}-x_N)^2+x_1^2+x_N^2$, which is positive, provided that at least one x_i is non-null.

Exercise 9.2 We verify that $\mathrm{A}\mathbf{q}_j = \lambda_j\mathbf{q}_j$. Computing the matrix-vector product $\mathbf{w} = \mathrm{A}\mathbf{q}_j$ and requiring that $\mathbf{w}$ is equal to the vector $\lambda_j\mathbf{q}_j$, we find:

$$\begin{cases} 2\sin(j\theta) - \sin(2j\theta) = 2(1-\cos(j\theta))\sin(j\theta), \\ -\sin(j(k-1)\theta) + 2\sin(jk\theta) - \sin(j(k+1)\theta) = 2(1-\cos(j\theta))\sin(kj\theta), \\ \hfill k = 2,\ldots,N-1 \\ 2\sin(Nj\theta) - \sin((N-1)j\theta) = 2(1-\cos(j\theta))\sin(Nj\theta). \end{cases}$$

The first equation is an identity since $\sin(2j\theta) = 2\sin(j\theta)\cos(j\theta)$. The other equations can be simplified in view of the sum-to-product formula

$$\sin((k-1)j\theta) + \sin((k+1)j\theta) = 2\sin(kj\theta)\cos(j\theta)$$

and noticing that $\sin((N+1)j\theta) = 0$ since $\theta = \pi/(N+1)$. Since A is symmetric and positive definite, its condition number is $K(\mathrm{A}) = \lambda_{max}/\lambda_{min}$, that is, $K(\mathrm{A}) = \lambda_N/\lambda_1 = (1-\cos(N\pi/(N+1)))/(1-\cos(\pi/(N+1)))$. By the identity $\cos(N\pi/(N+1)) = -\cos(\pi/(N+1))$ and by using the Taylor expansion of order 2 of the cosine function, we obtain $K(\mathrm{A}) \simeq (N+1)^2$, that is, $K(\mathrm{A}) \simeq h^{-2}$.

Exercise 9.3 We note that

$$u(\bar{x}+h) = u(\bar{x}) + hu'(\bar{x}) + \frac{h^2}{2}u''(\bar{x}) + \frac{h^3}{6}u'''(\bar{x}) + \frac{h^4}{24}u^{(4)}(\xi_+),$$
$$u(\bar{x}-h) = u(\bar{x}) - hu'(\bar{x}) + \frac{h^2}{2}u''(\bar{x}) - \frac{h^3}{6}u'''(\bar{x}) + \frac{h^4}{24}u^{(4)}(\xi_-),$$

where $\xi_+ \in (x, x+h)$ and $\xi_- \in (x-h, x)$. Summing the two expression we obtain

$$u(\bar{x}+h) + u(\bar{x}-h) = 2u(\bar{x}) + h^2u''(\bar{x}) + \frac{h^4}{24}(u^{(4)}(\xi_+) + u^{(4)}(\xi_-)),$$

which is the desired property.

Exercise 9.4 The matrix is tridiagonal with entries $a_{i,i-1} = -\mu/h^2 - \eta/(2h)$, $a_{ii} = 2\mu/h^2 + \sigma$, $a_{i,i+1} = -\mu/h^2 + \eta/(2h)$. The right-hand side, accounting for the boundary conditions, becomes $\mathbf{f} = (f(x_1) + \alpha(\mu/h^2 + \eta/(2h)), f(x_2), \ldots, f(x_{N-1}), f(x_N) + \beta(\mu/h^2 - \eta/(2h)))^T$.

Exercise 9.5 With the following instructions we compute the corresponding solutions to the three given values of h:

```
f=@(x) 1+sin(4*pi*x);
[x,uh11]=bvp(0,1,9,1,0,0.1,f,0,0);
[x,uh21]=bvp(0,1,19,1,0,0.1,f,0,0);
[x,uh41]=bvp(0,1,39,1,0,0.1,f,0,0);
```

We recall that $h = (b-a)/(N+1)$. Since we don't know the exact solution, to estimate the convergence order we compute an approximate solution on a very fine grid (for instance $h = 1/1000$), then we use this latter as a surrogate for the exact solution. We find:

```
[x,uhex]=bvp(0,1,999,1,0,0.1,f,0,0);
max(abs(uh11-uhex(1:100:end)))

ans =
   8.6782e-04
max(abs(uh21-uhex(1:50:end)))
ans =
   2.0422e-04
max(abs(uh41-uhex(1:25:end)))
ans =
   5.2789e-05
```

Halving h, the error is divided by 4, proving that the convergence order with respect to h is 2.

Exercise 9.6 We should modify the Program 9.1 in order to impose Neumann boundary conditions. In the Program 10.4 we show one possible implementation.

Program 10.4. neumann: numerical solution of a Neumann boundary-value problem

```
function [xh,uh]=neumann(a,b,N,mu,eta,sigma,bvpfun,...
                    ua,ub,varargin)
h = (b-a)/(N+1); xh = (linspace(a,b,N+2))';
hm = mu/h^2;  hd = eta/(2*h); e =ones(N+2,1);
A = spdiags([-hm*e-hd (2*hm+sigma)*e -hm*e+hd],...
     -1:1, N+2, N+2);
A(1,1)=3/(2*h); A(1,2)=-2/h; A(1,3)=1/(2*h); f(1)=ua;
A(N+2,N+2)=3/(2*h); A(N+2,N+1)=-2/h; A(N+2,N)=1/(2*h);
f =bvpfun(xh,varargin{:});  f(1)=ua; f(N+2)=ub;
uh = A\f;
```

Exercise 9.7 The trapezoidal integration formula, used on the two subintervals I_{j-1} and I_j, produces the following approximation

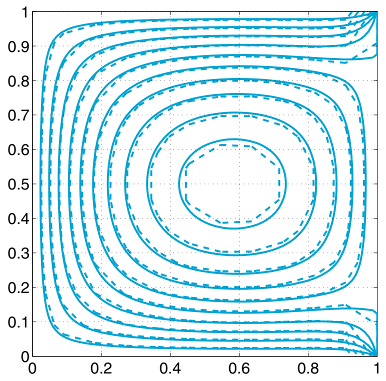


Figure 10.18. The contour lines of the computed temperature for $\varDelta_x = \varDelta_y = 1/10$ (*dashed lines*) and for $\varDelta_x = \varDelta_y = 1/80$ (*solid lines*)

$$\int\limits_{I_{j-1}\cup I_j}^{f} (x)\varphi_j(x)\ dx \simeq \frac{h}{2}f(x_j) + \frac{h}{2}f(x_j) = hf(x_j),$$

since $\varphi_j(x_i) = \delta_{ij}$ for any i, j. When $j = 1$ or $j = N$ we can proceed similarly, taking into account the Dirichlet boundary conditions. Thus, we obtain the same right-hand side of the finite difference system (9.14) up to the factor h.

Exercise 9.8 We have $\nabla\phi = (\partial\phi/\partial x, \partial\phi/\partial y)^T$ and therefore $\text{div}\nabla\phi = \partial^2\phi/\partial x^2 + \partial^2\phi/\partial y^2$, that is, the Laplacian of ϕ.

Exercise 9.9 To compute the temperature at the center of the plate, we solve the corresponding Poisson problem for various values of $\varDelta_x = \varDelta_y$, using the following instructions:

```
k=0; fun=@(x,y) 25+0*x+0*y;
bound=@(x,y) (x==1);
for N = [10,20,40,80,160]
[xh,yh,uh]=poissonfd(0,1,0,1,N,N,fun,bound);
k=k+1; uc(k) = uh(N/2+1,N/2+1);
end
```

The components of the vector `uc` are the values of the computed temperature at the center of the plate as the steplength h of the grid decreases. We have

```
uc
    2.0168    2.0616    2.0789    2.0859    2.0890
```

We can therefore conclude that at the center of the plate the temperature is about 2.08°C. In Figure 10.18 we show the contour lines of the temperature for two different values of h.

Exercise 9.10 For sake of simplicity we set $u_t = \partial u/\partial t$ and $u_x = \partial u/\partial x$. We multiply by u_t the equation (9.72) with $f \equiv 0$, integrate in space on (a, b) and use integration by parts on the second term:

$$\int_a^b u_{tt}(x,t)u_t(x,t)\mathrm{d}x + c\int_a^b u_x(x,t)u_{tx}(x,t)\mathrm{d}x - c[u_x(x,t)u_t(x,t)]_a^b = 0. \tag{10.7}$$

Now we integrate in time equation (10.7), from 0 up to t. By noticing that $u_{tt}u_t = \frac{1}{2}(u_t^2)_t$ and that $u_x u_{xt} = \frac{1}{2}(u_x^2)_t$, by applying the fundamental theorem of integral calculus and recalling the initial conditions (9.74) (that is $u_t(x,0) = v_0(x)$ and $u_x(x,0) = u_{0x}(x)$), we obtain

$$\int_a^b u_t^2(x,t)\mathrm{d}x + c\int_a^b u_x^2(x,t)\mathrm{d}x = \int_a^b v_0^2(x)\mathrm{d}x$$

$$+c\int_a^b u_{0x}^2(x)\mathrm{d}x + 2c\int_0^t \left(u_x(b,s)u_t(b,s) - u_x(a,s)u_t(a,s)\right)\mathrm{d}s.$$

On the other hand, by integrating by parts and applying the homogeneous Dirichlet boundary conditions for $t > 0$ and on the initial data we obtain

$$\int_0^t (u_x(b,s)u_t(b,s) - u_x(a,s)u_t(a,s))\mathrm{d}s = 0.$$

Then (9.83) follows.

Exercise 9.11 In view of definition (9.64) it is sufficient to verify that

$$\sum_{j=-\infty}^{\infty} |u_j^{n+1}|^2 \leq \sum_{j=-\infty}^{\infty} |u_j^n|^2. \tag{10.8}$$

In formula (9.62), let us move all terms to the left-hand side and then multiply by u_j^{n+1}. Owing to the identity $2(a-b)a = a^2 - b^2 + (a-b)^2$ we have

$$|u_j^{n+1}|^2 - |u_j^n|^2 + |u_j^{n+1} - u_j^n|^2 + \lambda a(u_{j+1}^{n+1} - u_{j-1}^{n+1})u_j^{n+1} = 0,$$

then, summing up on j and noticing that $\sum_{j=-\infty}^{\infty}(u_{j+1}^{n+1} - u_{j-1}^{n+1})u_j^{n+1} = 0$, we obtain

$$\sum_{j=-\infty}^{\infty} |u_j^{n+1}|^2 \leq \sum_{j=-\infty}^{\infty} |u_j^{n+1}|^2 + \sum_{j=-\infty}^{\infty} |u_j^{n+1} - u_j^n|^2 \leq \sum_{j=-\infty}^{\infty} |u_j^n|^2.$$

Exercise 9.12 The *upwind* scheme (9.59) can be rewritten in the simplified form

$$u_j^{n+1} = \begin{cases} (1-\lambda a)u_j^n + \lambda a u_{j-1}^n & \text{if } a > 0 \\ (1+\lambda a)u_j^n - \lambda a u_{j+1}^n & \text{if } a < 0. \end{cases}$$

Let us first consider the case $a > 0$. If the CFL condition is satisfied, then both coefficients $(1-\lambda a)$ and λa are positive and less than 1.
This fact implies that

$$\min\{u_{j-1}^n, u_j^n\} \leq u_j^{n+1} \leq \max\{u_{j-1}^n, u_j^n\}$$

and, by recursion on n, it holds

$$\inf_{l\in\mathbb{Z}}\{u_l^0\} \le u_j^{n+1} \le \sup_{l\in\mathbb{Z}}\{u_l^0\} \quad \forall n \ge 0,$$

from which the estimate (9.85) follows.
When $a < 0$, using again the CFL condition, both coefficients $(1 + \lambda a)$ and $-\lambda a$ are positive and less than 1. By proceeding as we did before, the estimate (9.85) follows also in this case.

Exercise 9.13 To numerically solve problem (9.47) we call the Program 10.5. Note that the exact solution is the travelling wave with velocity $a = 1$, that is $u(x,t) = 2\cos(4\pi(x-t)) + \sin(20\pi(x-t))$. Since the CFL number is fixed to 0.5, the discretization parameters Δx and Δt are related through the equation $\Delta t = CFL \cdot \Delta x$, thus we can arbitrarily choose only one of them.
In order to verify the accuracy of the scheme with respect to Δt we can use the following instructions:

```
xspan=[0,0.5];
tspan=[0,1];
a=1; cfl=0.5;
u0=@(x) 2*cos(4*pi*x)+sin(20*pi*x);
uex=@(x,t) 2*cos(4*pi*(x-t))+sin(20*pi*(x-t));
ul=@(t) 2*cos(4*pi*t)-sin(20*pi*t);
DT=[1.e-2,5.e-3,2.e-3,1.e-3,5.e-4,2.e-4,1.e-4];
e_lw=[]; e_up=[];
for deltat=DT
deltax=deltat*a/cfl;
[xx,tt,u_lw]=hyper(xspan,tspan,u0,ul,2,...
      cfl,deltax,deltat);
[xx,tt,u_up]=hyper(xspan,tspan,u0,ul,3,...
      cfl,deltax,deltat);
U=uex(xx,tt(end));
[Nx,Nt]=size(u_lw);
e_lw=[e_lw sqrt(deltax)*norm(u_lw(Nx,:)-U,2)];
e_up=[e_up sqrt(deltax)*norm(u_up(Nx,:)-U,2)];
end
p_lw=log(abs(e_lw(1:end-1)./e_lw(2:end)))./...
     log(DT(1:end-1)./DT(2:end))
p_up=log(abs(e_up(1:end-1)./e_up(2:end)))./...
     log(DT(1:end-1)./DT(2:end))
```

```
p_lw =
  0.1939    1.8626    2.0014    2.0040    2.0112    2.0239
p_up =
  0.2272    0.3604    0.5953    0.7659    0.8853    0.9475
```

By implementing a similar loop for the parameter Δx, we can verify the accuracy of the scheme with respect to the space discretization. Precisely, for Δx ranging from 10^{-4} to 10^{-2} we obtain

```
p_lw =
  1.8113    2.0235    2.0112    2.0045    2.0017    2.0007
p_up =
  0.3291    0.5617    0.7659    0.8742    0.9407    0.9734
```

Program 10.5. hyper: Lax-Friedrichs, Lax-Wendroff and upwind schemes

```
function [xh,th,uh]=hyper(xspan,tspan,u0,ul,...
                          scheme,cfl,deltax,deltat)
% HYPER solves hyperbolic scalar equations
% [XH,TH,UH]=HYPER(XSPAN,TSPAN,U0,UL,SCHEME,CFL,...
%                  DELTAX,DELTAT)
% solves the hyperbolic scalar equation
%       DU/DT+ A * DU/DX=0
% in (XSPAN(1),XSPAN(2))x(TSPAN(1),TSPAN(2))
% with A>0, initial condition  U(X,0)=U0(X) and
% boundary condition U(T)=UL(T) given at XSPAN(1)
% with several finite difference schemes.
% scheme = 1 Lax - Friedrichs
%          2 Lax - Wendroff
%          3 Upwind
% The propagation velocity 'a' is not required as
% input of the function, since it can be derived
% from CFL = A * DELTAT / DELTAX
% Output: XH is the vector of space nodes
% TH  is the vector of time nodes
% UH is a matrix containing the computed solution
% UH(n,:) contains the solution at time TT(n)
% U0 and UL can be either inline, anonymous
% functions or functions defined by M-file.
Nt=(tspan(2)-tspan(1))/deltat+1;
th=linspace(tspan(1),tspan(2),Nt);
Nx=(xspan(2)-xspan(1))/deltax+1;
xh=linspace(xspan(1),xspan(2),Nx);
u=zeros(Nt,Nx); cfl2=cfl*0.5; cfl21=1-cfl^2;
cflp1=cfl+1; cflm1=cfl-1; uh(1,:)=u0(xh);
for n=1:Nt-1
 uh(n+1,1)=ul(th(n+1));
 if scheme == 1
% Lax Friedrichs
    for j=2:Nx-1
      uh(n+1,j)=0.5*(-cflm1*uh(n,j+1)+cflp1*uh(n,j-1));
    end
    j=Nx;
    uh(n+1,j)=0.5*(-cflm1*(2*uh(n,j)-uh(n,j-1))+...
        cflp1*uh(n,j-1));
 elseif scheme == 2
% Lax Wendroff
    for j=2:Nx-1
     uh(n+1,j)=cfl21*uh(n,j)+...
         cfl2*(cflm1*uh(n,j+1)+cflp1*uh(n,j-1));
    end
    j=Nx;
    uh(n+1,j)=cfl21*uh(n,j)+...
    cfl2*(cflm1*(2*uh(n,j)-uh(n,j-1))+cflp1*uh(n,j-1));
 elseif scheme ==3
% Upwind
   for j=2:Nx
        uh(n+1,j)=-cflm1*uh(n,j)+cfl*uh(n,j-1);
    end
 end
end
```

Exercise 9.14 The exact solution is the sum of two simple harmonics, the former with low frequency and the latter with high frequency. If we choose $\Delta t = 5 \cdot 10^{-2}$, since $a = 1$ and CFL=0.8, we have $\Delta x = 6.25e - 3$ and the phase angles associated to the harmonics are $\phi_{k_1} = 4\pi \cdot 6.25e - 3 \simeq 0.078$ and $\phi_{k_2} = 20\pi \cdot 6.25e - 3 \simeq 0.393$, respectively. By inspecting Figure 9.18 we note that the upwind scheme is more dissipative than Lax-Wendroff's. This fact is confirmed by the behavior of dissipation coefficients (see the right graph at the bottom of Figure 9.14). Indeed, when we take into account instances of ϕ_k corresponding to the given harmonics, the curve relative to the Lax-Wendroff scheme is nearer to the constant 1 than the curve associated to the upwind scheme.

For what concerns the dispersion coefficient, we see from Figure 9.18 that the Lax-Wendroff scheme features a phase delay, while the *upwind* scheme presents a light phase advance. The right graph at the bottom of Figure 9.15 confirms this conclusion. Moreover we can observe that the phase delay of the Lax-Wendroff scheme is larger than the phase advance of the upwind scheme.

References

[ABB$^+$99] Anderson E., Bai Z., Bischof C., Blackford S., Demmel J., Dongarra J., Croz J. D., Greenbaum A., Hammarling S., McKenney A., and Sorensen D. (1999) *LAPACK User's Guide.* 3rd edition. SIAM, Philadelphia.

[Ada90] Adair R. (1990) *The Physics of Baseball.* Harper and Row, New York.

[Arn73] Arnold V. (1973) *Ordinary Differential Equations.* The MIT Press, Cambridge.

[Atk89] Atkinson K. (1989) *An Introduction to Numerical Analysis.* 2nd edition. John Wiley & Sons Inc., New York.

[Att11] Attaway S. (2011) *MATLAB: A Practical Introduction to Programming and Problem Solving,* 2nd edition. Butterworth-Heinemann. Elsevier, Waltham, MA.

[Axe94] Axelsson O. (1994) *Iterative Solution Methods.* Cambridge University Press, Cambridge.

[BB96] Brassard G. and Bratley P. (1996) *Fundamentals of Algorithmics.* Prentice Hall Inc., Englewood Cliffs, NJ.

[BDF$^+$10] Bomze I., Demyanov V., Fletcher R., Terlaky T., and Polik I. (2010) *Nonlinear Optimization*, volume 1989 of *Lecture Notes in Mathematics.* Springer. Lectures given at the C.I.M.E. Summer School held in Cetraro, July 2007. Edited by G. Di Pillo and F. Schoen.

[Bec71] Beckmann P. (1971) *A history of π.* 2nd edtion. The Golem Press, Boulder, Colorado.

[Ber82] Bertsekas D. (1982) *Constrained optimization and Lagrange multipliers methods.* Academic Press, Inc., San Diego, CA.

[BGL05] Benzi M., Golub G., and Liesen J. (2005) Numerical solution of saddle point problems. *Acta Numer.* 14: 1–137.

[BM92] Bernardi C. and Maday Y. (1992) *Approximations Spectrales des Problémes aux Limites Elliptiques.* Springer-Verlag, Paris.

[Bom10] Bomze M. (2010) *Global Optimization: A Quadratic Programming Perspective*, volume 1989 of *Lecture Notes in Mathematics*, chapter 3, pages 1–53. Springer. Lectures given at the C.I.M.E.

Summer School held in Cetraro, July 2007. Edited by G. Di Pillo and F. Schoen.

[Bra97] Braess D. (1997) *Finite Elements: Theory, Fast Solvers and Applications in Solid Mechanics.* Cambridge University Press, Cambridge.

[Bre02] Brent R. (2002) *Algorithms for minimization without derivatives.* Dover Publications Inc., Mineola, NY. Reprint of the 1973 original [Prentice-Hall, Inc., Englewood Cliffs, NJ].

[BS01] Babuska I. and Strouboulis T. (2001) *The Finite Element Method and its Reliability.* Numerical Mathematics and Scientific Computation. The Clarendon Press Oxford University Press, New York.

[BT04] Berrut J.-P. and Trefethen L.-N. (2004) Barycentric Lagrange Interpolation. *SIAM Review* 46(3): 501–517.

[But87] Butcher J. (1987) *The Numerical Analysis of Ordinary Differential Equations: Runge-Kutta and General Linear Methods.* Wiley, Chichester.

[CFL28] Courant R., Friedrichs K., and Lewy H. (1928) Über die partiellen Differenzengleichungen der mathematischen Physik. *Math. Ann.* 100(1): 32–74.

[CHQZ06] Canuto C., Hussaini M. Y., Quarteroni A., and Zang T. A. (2006) *Spectral Methods: Fundamentals in Single Domains.* Scientific Computation. Springer-Verlag, Berlin.

[CHQZ07] Canuto C., Hussaini M. Y., Quarteroni A., and Zang T. A. (2007) *Spectral Methods. Evolution to Complex Geometries and Applications to Fluid Dynamics.* Scientific Computation. Springer, Heidelberg.

[CL96a] Coleman T. and Li Y. (1996) An interior trust region approach for nonlinear minimization subject to bounds. *SIAM J. Optim.* 6(2): 418–445.

[CL96b] Coleman T. and Li Y. (1996) A reflective Newton method for minimizing a quadratic function subject to bounds on some of the variables. *SIAM J. Optim.* 6(4): 1040–1058.

[CLW69] Carnahan B., Luther H., and Wilkes J. (1969) *Applied Numerical Methods.* John Wiley & Sons, Inc., New York.

[Dav63] Davis P. (1963) *Interpolation and Approximation.* Blaisdell Publishing Co. Ginn and Co. New York-Toronto-London, New York.

[dB01] de Boor C. (2001) *A practical guide to splines.* Applied Mathematical Sciences. Springer-Verlag, New York.

[DD99] Davis T. and Duff I. (1999) A combined unifrontal/multifrontal method for unsymmetric sparse matrices. *ACM Transactions on Mathematical Software* 25(1): 1–20.

[Dem97] Demmel J. (1997) *Applied Numerical Linear Algebra.* SIAM, Philadelphia.

[Deu04] Deuflhard P. (2004) *Newton Methods for Nonlinear Problems. Affine Invariance and Adaptive Algorithms.* Springer Series in Computational Mathematics. Springer-Verlag, Berlin.

[Die93] Dierckx P. (1993) *Curve and Surface Fitting with Splines.* Monographs on Numerical Analysis. The Clarendon Press Oxford University Press, New York.

[DL92] DeVore R. and Lucier B. (1992) Wavelets. In *Acta numerica, 1992*, pages 1–56. Cambridge Univ. Press, Cambridge.

[DR75] Davis P. and Rabinowitz P. (1975) *Methods of Numerical Integration.* Academic Press, New York.

[dV89] der Vorst H. V. (1989) High Performance Preconditioning. *SIAM J. Sci. Stat. Comput.* 10: 1174–1185.

[EBH08] Eaton J., Bateman D., and Hauberg S. (2008) *GNU Octave Manual Version 3.* Network Theory Ltd., Bristol.

[EEHJ96] Eriksson K., Estep D., Hansbo P., and Johnson C. (1996) *Computational Differential Equations.* Cambridge Univ. Press, Cambridge.

[EKM05] Etter D., Kuncicky D., and Moore H. (2005) *Introduction to MATLAB 7.* Prentice Hall, Englewood Cliffs.

[Eva98] Evans L. (1998) *Partial Differential Equations.* American Mathematical Society, Providence.

[Fle10] Fletcher R. (2010) *The Sequential Quadratic Programming Method*, volume 1989 of *Lecture Notes in Mathematics*, chapter 3, pages 165–214. Springer. Lectures given at the C.I.M.E. Summer School held in Cetraro, July 2007. Edited by G. Di Pillo and F. Schoen.

[Fun92] Funaro D. (1992) *Polynomial Approximation of Differential Equations.* Springer-Verlag, Berlin Heidelberg.

[Gau97] Gautschi W. (1997) *Numerical Analysis. An Introduction.* Birkhäuser Boston Inc., Boston, MA.

[Gea71] Gear C. (1971) *Numerical Initial Value Problems in Ordinary Differential Equations.* Prentice-Hall, Upper Saddle River NJ.

[GI04] George A. and Ikramov K. (2004) Gaussian elimination is stable for the inverse of a diagonally dominant matrix. *Math. Comp.* 73(246): 653–657.

[GL96] Golub G. and Loan C. V. (1996) *Matrix Computations.* 3rd edition. The John Hopkins Univ. Press, Baltimore, MD.

[GM72] Gill P. and Murray W. (1972) Quasi-Newton methods for unconstrained optimization. *J. Inst. Math. Appl.* 9: 91–108.

[GN06] Giordano N. and Nakanishi H. (2006) *Computational Physics.* 2nd edition. Prentice-Hall, Upper Saddle River NJ.

[GOT05] Gould N., Orban D., and Toint P. (2005) Numerical methods for large-scale nonlinear optimization. *Acta Numerica* 14: 299–361.

[GR96] Godlewski E. and Raviart P.-A. (1996) *Hyperbolic Systems of Conservations Laws.* Springer-Verlag, New York.

[Hac85] Hackbusch W. (1985) *Multigrid Methods and Applications.* Springer Series in Computational Mathematics. Springer-Verlag, Berlin.

[Hac94] Hackbusch W. (1994) *Iterative Solution of Large Sparse Systems of Equations.* Applied Mathematical Sciences. Springer-Verlag, New York.

[Hen79] Henrici P. (1979) Barycentric formulas for interpolating trigonometric polynomials and their conjugate. *Numer. Math.* 33: 225–234.

[Hes98] Hesthaven J. (1998) From electrostatics to almost optimal nodal sets for polynomial interpolation in a simplex. *SIAM J. Numer. Anal.* 35(2): 655–676.

[HH05] Higham D. and Higham N. (2005) *MATLAB Guide.* 2nd edition. SIAM Publications, Philadelphia, PA.

[Hig02] Higham N. (2002) *Accuracy and Stability of Numerical Algorithms.* 2nd edition. SIAM Publications, Philadelphia, PA.

[Hig04] Higham N.-J. (2004) The numerical stability of barycentric lagrange interpolation. *IMA J. Numer. Anal.* 24(4): 547–556.

[Hir88] Hirsh C. (1988) *Numerical Computation of Internal and External Flows.* John Wiley and Sons, Chichester.

[HLR06] Hunt B., Lipsman R., and Rosenberg J. (2006) *A guide to MATLAB. For Beginners and Experienced Users.* 2nd edition. Cambridge University Press, Cambridge.

[IK66] Isaacson E. and Keller H. (1966) *Analysis of Numerical Methods.* Wiley, New York.

[Joh90] Johnson C. (1990) *Numerical Solution of Partial Diffferential Equations by the Finite Element Method.* Cambridge University Press, Cambridge.

[JS96] Jr J. D. and Schnabel R. (1996) *Numerical methods for unconstrained optimization and nonlinear equations*, volume 16 of *Classics in Applied Mathematics.* Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA.

[Krö98] Kröner D. (1998) Finite Volume Schemes in Multidimensions. In *Numerical analysis 1997 (Dundee)*, Pitman Res. Notes Math. Ser., pages 179–192. Longman, Harlow.

[KS99] Karniadakis G. and Sherwin S. (1999) *Spectral/hp Element Methods for CFD.* Oxford University Press, New York.

[KW08] Kalos M. and Whitlock P. (2008) *Monte Carlo Methods,* 2nd ed. John Wiley & Sons.

[Lam91] Lambert J. (1991) *Numerical Methods for Ordinary Differential Systems.* John Wiley and Sons, Chichester.

[Lan03] Langtangen H. (2003) *Advanced Topics in Computational Partial Differential Equations: Numerical Methods and Diffpack Programming.* Springer-Verlag, Berlin Heidelberg.

[LeV02] LeVeque R. (2002) *Finite Volume Methods for Hyperbolic Problems.* Cambridge University Press, Cambridge.

[LRWW99] Lagarias J., Reeds J., Wright M., and Wright P. (1999) Convergence properties of the Nelder-Mead simplex method in low dimensions. *SIAM J. Optim.* 9(1): 112–147.

[Mei67] Meinardus G. (1967) *Approximation of Functions: Theory and Numerical Methods.* Springer Tracts in Natural Philosophy. Springer-Verlag New York, Inc., New York.

[MH03] Marchand P. and Holland O. (2003) *Graphics and GUIs with MATLAB.* 3rd edition. Chapman & Hall/CRC, London, New York.

[Mun07] Munson T. (2007) Mesh shape-quality optimization using the inverse mean-ratio metric. *Math. Program.* 110(3, Ser. A): 561–590.

[Nat65] Natanson I. (1965) *Constructive Function Theory. Vol. III. Interpolation and approximation quadratures.* Frederick Ungar Publishing Co., New York.

[NM65] Nelder J. and Mead R. (1965) A simplex method for function minimization. *The Computer Journal* 7: 308–313.

[Noc92] Nocedal J. (1992) Theory of algorithms for unconstrained optimization. In *Acta numerica, 1992*, pages 199–242. Cambridge Univ. Press, Cambridge.

[NW06] Nocedal J. and Wright S. (2006) *Numerical optimization.* Springer Series in Operations Research and Financial Engineering. Springer, New York, second edition.

[OR70] Ortega J. and Rheinboldt W. (1970) *Iterative Solution of Nonlinear Equations in Several Variables.* Academic Press, New York, London.

[Pal08] Palm W. (2008) *A Concise Introduction to Matlab.* McGraw-Hill, New York.

[Pan92] Pan V. (1992) Complexity of Computations with Matrices and Polynomials. *SIAM Review* 34(2): 225–262.

[Pap87] Papoulis A. (1987) *Probability, Random Variables, and Stochastic Processes.* McGraw-Hill, New York.

[PBP02] Prautzsch H., Boehm W., and Paluszny M. (2002) *Bezier and B-Spline Techniques.* Mathematics and Visualization. Springer-Verlag, Berlin.

[PdDKÜK83] Piessens R., de Doncker-Kapenga E., Überhuber C., and Kahaner D. (1983) *QUADPACK: A Subroutine Package for Automatic Integration.* Springer Series in Computational Mathematics. Springer-Verlag, Berlin.

[Pra06] Pratap R. (2006) *Getting Started with MATLAB 7: A Quick Introduction for Scientists and Engineers.* Oxford University Press, New York.

[QSS07] Quarteroni A., Sacco R., and Saleri F. (2007) *Numerical Mathematics.* 2nd edition. Texts in Applied Mathematics. Springer-Verlag, Berlin.

[Qua13] Quarteroni A. (2013) *Numerical Models for Differential Problems.* Series: MS&A , Vol. 2*, 2nd edition.* Springer-Verlag, Milano.

[QV94] Quarteroni A. and Valli A. (1994) *Numerical Approximation of Partial Differential Equations.* Springer-Verlag, Berlin.

[Ros61] Rosenbrock H. (1960/1961) An automatic method for finding the greatest or least value of a function. *Comput. J.* 3: 175–184.

[RR01] Ralston A. and Rabinowitz P. (2001) *A First Course in Numerical Analysis.* 2nd edition. Dover Publications Inc., Mineola, NY.

[Saa92] Saad Y. (1992) *Numerical Methods for Large Eigenvalue Problems.* Manchester University Press, Manchester; Halsted Press (John Wiley & Sons, Inc.), Manchester; New York.

[Saa03] Saad Y. (2003) *Iterative Methods for Sparse Linear Systems.* 2nd edition. SIAM publications, Philadelphia, PA.

[Sal08] Salsa S. (2008) *Partial Differential Equations in Action - From Modelling to Theory.* Springer, Milan.

[SM03] Süli E. and Mayers D. (2003) *An Introduction to Numerical Analysis.* Cambridge University Press, Cambridge.

[SR97] Shampine L. and Reichelt M. (1997) The MATLAB ODE suite. *SIAM J. Sci. Comput.* 18(1): 1–22.

[SSB85] Shultz G., Schnabel R., and Byrd R. (1985) A family of trust-region-based algorithms for unconstrained minimization with strong global convergence properties. *SIAM J. Numer. Anal.* 22(1): 47–67.

[Ste83] Steihaug T. (1983) The conjugate gradient method and trust regions in large scale optimization. *SIAM J. Numer. Anal.* 20(3): 626–637.

[Str07] Stratton J. (2007) *Electromagnetic Theory.* Wiley-IEEE Press, Hoboken, New Jersey.

[SY06] Sun W. and Yuan Y.-X. (2006) *Optimization theory and methods*, volume 1 of *Springer Optimization and Its Applications.* Springer, New York. Nonlinear programming.

[Ter10] Terlaky T. (2010) *Interior Point Methods for Nonlinear Optimization*, volume 1989 of *Lecture Notes in Mathematics*, chapter 3, pages 215–276. Springer. Lectures given at the C.I.M.E. Summer School held in Cetraro, July 2007. Edited by G. Di Pillo and F. Schoen.

[TW98] Tveito A. and Winther R. (1998) *Introduction to Partial Differential Equations. A Computational Approach.* Springer-Verlag, Berlin Heidelberg.

[Übe97] Überhuber C. (1997) *Numerical Computation: Methods, Software, and Analysis.* Springer-Verlag, Berlin.

[Urb02] Urban K. (2002) *Wavelets in Numerical Simulation.* Lecture Notes in Computational Science and Engineering. Springer-Verlag, Berlin.

[vdV03] van der Vorst H. (2003) *Iterative Krylov Methods for Large Linear Systems.* Cambridge Monographs on Applied and Computational Mathematics. Cambridge University Press, Cambridge.

[VGCN05] Valorani M., Goussis D., Creta F., and Najm H. (2005) Higher order corrections in the approximation of low-dimensional manifolds and the construction of simplified problems with the CSP method. *J. Comput. Phys.* 209(2): 754–786.

[Wes04] Wesseling P. (2004) *An Introduction to Multigrid Methods.* R.T. Edwards, Inc., Philadelphia.

[Wil88] Wilkinson J. (1988) *The Algebraic Eigenvalue Problem.* Monographs on Numerical Analysis. The Clarendon Press Oxford University Press, New York.

[Zha99] Zhang F. (1999) *Matrix theory.* Universitext. Springer-Verlag, New York.

Index

Editorial Policy

1. Textbooks on topics in the field of computational science and engineering will be considered. They should be written for courses in CSE education. Both graduate and undergraduate textbooks will be published in TCSE. Multidisciplinary topics and multidisciplinary teams of authors are especially welcome.
2. Format: Only works in English will be considered. For evaluation purposes, manuscripts may be submitted in print or electronic form, in the latter case, preferably as pdf- or zipped ps-files. Authors are requested to use the LaTeX style files available from Springer at: http://www.springer.com/authors/book+authors/helpdesk?SGWID=0-1723113-12-971304-0 (for monographs, textbooks and similar)
   Electronic material can be included if appropriate. Please contact the publisher.
3. Those considering a book which might be suitable for the series are strongly advised to contact the publisher or the series editors at an early stage.

General Remarks

Careful preparation of manuscripts will help keep production time short and ensure a satisfactory appearance of the finished book.

The following terms and conditions hold:

Regarding free copies and royalties, the standard terms for Springer mathematics textbooks hold. Please write to martin.peters@springer.com for details.

Authors are entitled to purchase further copies of their book and other Springer books for their personal use, at a discount of 33.3% directly from Springer-Verlag.

Series Editors