UNITEXT 121

Alfio Quarteroni · Paola Gervasio

A Primer on Mathematical Modelling

UNITEXT – La Matematica per il 3+2

Volume 121

Editor-in-Chief
Alfio Quarteroni, Politecnico di Milano, Milan, Italy; École Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland

Series Editors
Luigi Ambrosio, Scuola Normale Superiore, Pisa, Italy
Paolo Biscari, Politecnico di Milano, Milan, Italy
Ciro Ciliberto, Università di Roma "Tor Vergata", Rome, Italy
Camillo De Lellis, Institute for Advanced Study, Princeton, NJ, USA
Massimiliano Gubinelli, Hausdorff Center for Mathematics, Rheinische Friedrich-Wilhelms-Universität, Bonn, Germany
Victor Panaretos, Institute of Mathematics, EPFL, Lausanne, Switzerland

The UNITEXT – La Matematica per il 3+2 series is designed for undergraduate and graduate academic courses, and also includes advanced textbooks at a research level.

Originally released in Italian, the series now publishes textbooks in English addressed to students in mathematics worldwide.

Some of the most successful books in the series have evolved through several editions, adapting to the evolution of teaching curricula.

Submissions must include at least 3 sample chapters, a table of contents, and a preface outlining the aims and scope of the book, how the book fits in with the current literature, and which courses the book is suitable for.

For any further information, please contact the Editor at Springer:
francesca.bonadei@springer.com

THE SERIES IS INDEXED IN SCOPUS

More information about this subseries at http://www.springer.com/series/5418

Alfio Quarteroni • Paola Gervasio

A Primer on Mathematical Modelling

Springer

Alfio Quarteroni
MOX, Department of Mathematics
Politecnico di Milano
Milano, Italy

Ecole Polytechnique
Fédérale de Lausanne (EPFL)
Lausanne, Switzerland

Paola Gervasio
DICATAM
University of Brescia
Brescia, Italy

Translated by
Simon G. Chiossi
Departamento de Matemática Aplicada
Universidade Federal Fluminense
Niterói, Brazil

ISSN 2038-5714 ISSN 2532-3318 (electronic)
UNITEXT
ISSN 2038-5722 ISSN 2038-5757 (electronic)
La Matematica per il 3+2
ISBN 978-3-030-44540-9 ISBN 978-3-030-44541-6 (eBook)
https://doi.org/10.1007/978-3-030-44541-6

Based on a translation from the Italian language edition: I delfini delle Eolie, i battiti del cuore, i motori di ricerca by Alfio Quarteroni, Paola Gervasio © Zanichelli 2019 All rights Reserved.
© Springer Nature Switzerland AG 2020
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors, and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Cover Illustration: Dolphins. Photo by Wolfgang Zimmel, 2017.

This Springer imprint is published by the registered company Springer Nature Switzerland AG.
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

To Fausto

Preface

How does one make money by developing apps?

How can we assess how profitable a small company producing teenager fashion is?

How does Google actually work when we ask it a question online, even the most baffling one?

How does one plan seasonal fishing in order to achieve the right balance between catch and stock reproduction?

Do these seem to you weird questions with no logical connection?

Well, they do have something in common, and that is mathematics, or rather the mathematical models allowing to give clear, precise, and exhaustive answers to these and many other questions. Mathematical models clarify to us that utmost miracle produced by a beating heart, by which every second of our lives the (approximately) one hundred thousand billion cells in our body are supplied with oxygen and nutrients vehicled through the circulatory system.

Yes, mathematical models do exactly that: they represent every aspect of the real world, even the most complex ones, using the language and tools of mathematics. By doing so they help us comprehend these aspects, simulate what will happen, verify it, and even make predictions.

This book aims to help understand how to derive a mathematical model, how it works, what potential it has. We will do this by means of examples (those mentioned above), but with the intent of describing the processes underpinning a mathematical model in its generality. In this way we wish to encourage you, the reader, to model other problems and become more aware of the power of representation of mathematics.

Through models we want to introduce you to the beauty of Scientific Computing. This is a subject that bridges between theory and computers, and allows to concretely solve problems that it would never be possible to solve by hand. This book is addressed to students interested in learning how to construct and apply mathematical models, and requires no background either in Calculus or in Linear Algebra.

Are you ready to start this journey between mathematics, algorithms, programming codes and computer? If so, then good luck and . . . good heartbeats!

Milano, Italy — Alfio Quarteroni
Brescia, Italy — Paola Gervasio
September 2020

About the Book

Tips for Further Reading

Each chapter opens with a small table summarising the main mathematical tools and new concepts that will be acquired, together with the prior knowledge required to understand the material.

Chapter 3, regarding Octave,[1] is not strictly necessary to any other. We organised the work so that the reader can follow the presentation even without any knowledge of coding. On the other hand, Octave can boost the problem-solving skills of those who do know how to code. For this reason we made available a number of programs (also known as functions and scripts), that allow to examine the numerical results and plots of the text just by typing simple commands provided along the way.

Readers choosing to make use of Octave can carry out the work of Chaps. 4, 5 and 6 in two different ways. The first way is to simply launch the functions and scripts we have provided (full codes were prepared by us for this purpose, and can be found at the end of each chapter, see also the *"Octave functions and scripts"* section). The second way involves a deeper understanding of our programs, following up on the study of Chap. 3.

Octave Functions and Scripts

A number of Octave functions and scripts have been made available to simplify the part of the work regarding testing. These files can be found at the end of each chapter and are collected in the archive `QG_functions_scripts.zip`, which may be downloaded from the webpage

http://paola-gervasio.unibs.it/quarteroni-gervasio/

In order to execute functions and scripts, Octave must be installed on your computer. Octave is a freely distributed software for Windows, MacOS and Linux operating systems, available at www.octave.org. The website www.octave.org

[1] John W. Eaton, David Bateman, Søren Hauberg, Rik Wehbring (2019). GNU Octave version 5.1.0 manual: a high-level interactive language for numerical computations. URL: https://www.gnu.org/software/octave/doc/v5.1.0/

provides installation instructions, which may be complemented by our own tips at http://paola-gervasio.unibs.it/quarteroni-gervasio/.

After having downloaded the file `QG_functions_scripts.zip` on your computer, you should unzip it, then launch Octave and modify the "Current Directory" (in the upper bar of the Octave window, see Fig. 3.1) to `QG_functions_scripts`.

All Octave instructions provided in the book can be executed without prior knowledge of its programming language. It suffices to launch Octave and type what we will indicate inside the "Command Window" (the empty window with the symbol »). With the help provided in Chap. 3, the most intrepid readers may rewrite by themselves all functions and scripts, and then compare them with our programs.

Contents

1 A Warm-Up to Models 1
1.1 Mathematical Modelling 2
1.2 Numerical Modelling and Scientific Computing 2
1.3 Data–Model – Results 4
1.3.1 Different Physical Problems, the Same Mathematical Model (a Subtle, Yet Crucial, Difference) 4
1.3.2 One Physical Problem, Many Mathematical Models 6
1.4 Errors, Errors... But No Panic 8
1.5 There Is Error and Error 12
1.6 What We Have Learnt 14

2 Getting Started 15
2.1 When the Solution to the Mathematical Model Is Not Known 16
2.1.1 Quadrature Formulas, or How to Compute Areas Approximately 18
2.1.2 The Errors in the Quadrature Formula 22
2.1.3 The Numerical Solution of Problem "Heights" 23
2.2 Vectors and Matrices to Handle Complexity 25
2.2.1 Vectors and Matrices 27
2.2.2 The 2×2 Linear System in Matrix Form 31
2.2.3 The Cramer Method for 2×2 Systems 31
2.2.4 The Gauss Elimination Method (GEM) for a 2×2 System 32
2.2.5 From 2 to 3 33
2.2.6 The 3×3 Linear System in Matrix Form 35
2.2.7 The Cramer Method for a 3×3 System 36
2.2.8 *for* Loops in Algorithms 39
2.2.9 Variable Assignment in an Algorithm 40
2.2.10 The Gauss Elimination Method (GEM) for a 3×3 System 41
2.2.11 We Reached 100! 45
2.2.12 How Many Operations Must We Do? 46
2.2.13 The Computer Comes into Play 47
2.2.14 GEM, Too, Adapts to the Computer 48
2.3 What We Have Learnt 50

3 Reckoning with the Computer 53
3.1 Programming with Octave 54
3.1.1 Arithmetic Operations and Variables 55
3.1.2 Mathematical Functions and Their Graphical Representation 57
3.1.3 Vectors and Matrices 59
3.1.4 Programming Units: Script and User-Defined Functions 71
3.1.5 Fundamental Statements 75
3.1.6 Tidbits of Graphics 83
3.2 How Computers Represent Numbers 86
3.2.1 Floating-Point Numbers 88
3.2.2 Roundoff Errors 90
3.3 What We Have Learnt 95
3.4 Script of This Chapter 95

4 A Virtual Surfer for the Web 97
4.1 Surfing with Google 97
4.2 Understanding the Problem 98
4.2.1 The PageRank 100
4.3 Simplify in Order to Model 101
4.3.1 The Directed Graph as a Model of the Web 101
4.3.2 The Adjacency Matrix of a Graph 102
4.4 From the Adjacency Matrix A to the Google Matrix G 103
4.5 The Mathematical Model 106
4.6 The Numerical Model: How Do We Solve the System $\mathbf{p} = G\mathbf{p}$? 108
4.7 The Algorithm for Computing PageRanks 111
4.7.1 While Loops in Algorithms 111
4.8 What We have Learnt 114
4.9 A Bit of History and a Glimpse Beyond 114
4.10 List of Functions and Scripts of the Chapter 116

5 A Network of Capillaries 119
5.1 Heart, Arteries, Veins and Capillaries 120
5.2 Understanding the Problem 123
5.2.1 The Beat That Keeps Us Alive 123
5.2.2 Fluid Flow and Viscosity 124
5.2.3 The Laminar Flow of a Viscous Fluid in a Cylindrical Pipe 126
5.3 Simplify in Order to Model 127
5.4 The Model for One Capillary 127
5.5 Blood Flow in a Small Capillary Bed 130
5.5.1 The Abstraction: From the Capillary Bed to the Graph 131
5.5.2 To Compute the Speeds We Need the Pressures 132
5.5.3 To Compute the Pressures We Need the Balance Equations 133
5.5.4 A System of Balance Equations 134
5.5.5 From the Matrix G of the Graph to the Matrix A of the System 136

5.5.6 Selection Blocks in Algorithms 138
5.5.7 From the Mathematical Model to the Solution: GEM 138
5.5.8 Volumetric Flow Rates and Velocities in Each Capillary 139
5.6 A More Complex Capillary Bed 142
5.7 What We Have Learnt 147
5.8 What Lies Beyond 147
5.9 List of Functions and Scripts of the Chapter 148

6 Predators and Preys in the Maths Ocean 153
6.1 Fishing in the Adriatic Sea 154
6.2 Simplify in Order to Model 155
6.3 Understanding the Problem 157
6.4 The Dynamics of One Population 158
6.4.1 The Natural Dynamics 158
6.4.2 The Interaction with the Environment: Constant Harvest in Time 162
6.4.3 The Interaction with the Environment: Time-Dependent Harvest (the Model) 164
6.4.4 A Numerical Method to Approximate First Derivatives 165
6.4.5 How to Represent Errors 169
6.4.6 The Euler Method for Approximating a Differential Equation 171
6.4.7 Error of the Euler Method 175
6.4.8 The Interaction With the Environment: Time-Dependent Harvest (the Numerical Solution) 176
6.4.9 An Even More Realistic Model: The Competition Within the Population 179
6.5 The Dynamics of Two Populations 182
6.5.1 The Prey-Predator System (the Model) 183
6.5.2 Numerical Approximation of a System of Differential Equations 184
6.5.3 The Prey-Predator System (the Numerical Solution) 185
6.5.4 An Even More Realistic Model: The Prey-Predator System with Harvest 188
6.5.5 The Prey-Predator Model with Harvest: The Validation 190
6.6 What We Have Learnt 191
6.7 Looking Beyond 192
6.8 List of Functions and Scripts of the Chapter 194

7 Take-Home Message 201

8 Solutions to Exercises 203
8.1 Exercises of Chapter 2 203
8.2 Exercises of Chapter 3 211
8.3 Exercises of Chapter 4 216
8.4 Exercises of Chapter 5 220
8.5 Exercises of Chapter 6 226

Index 235

Main Symbols

Symbol	Meaning	Page
$\simeq$	approximately equal to	10
$h \to 0$	h tends to zero	12
$x_n(h) \to x_m$ as $h \to 0$	$x_n(h)$ tends to x_m as h tends to 0	12
$\approx$	behaves like	13
$k = 1, \ldots, M$	for k ranging from 1 to M	19
$\mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$	two-dimensional column vector	27
$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$	2×2 matrix	28
$n! = 1 \cdot 2 \cdot 3 \cdots (n-1) \cdot n$	factorial of an integer number $n > 0$	46
$\prod_{k=1}^{n} a_k = a_1 \cdot a_2 \cdots a_n$	product of n numbers	46
$\sum_{k=1}^{M} a_k = a_1 + a_2 + \ldots + a_M$	sum of M numbers	20

Relevant concepts and remarks

To highlight pieces of text and formulas that are relevant.

Mathematical models

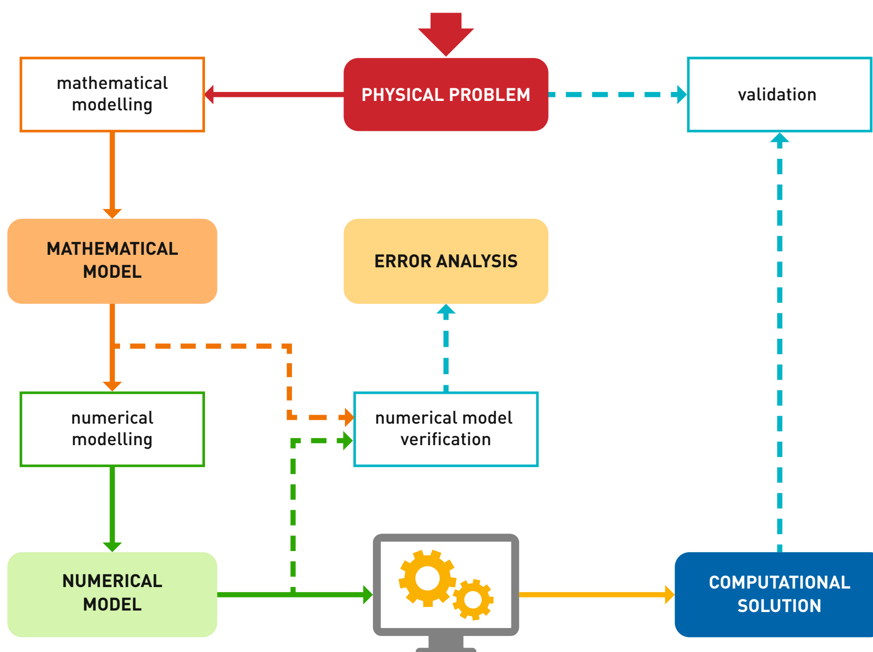
To highlight the mathematical models.

Exercises

To highlight the text of the exercises inside Chaps. 3–6 as well as to suggest solving the exercises of Chap. 8.

A Warm-Up to Models

1



Boxes with white background represent processes, while coloured boxes represent either the input or the output of those processes.

INGREDIENTS	Definition of function of one variable, average velocity.
WHAT WE LEARN	Mathematical model, numerical model, errors, validation, verification.
PREREQUISITES	None.

© Springer Nature Switzerland AG 2020

A. Quarteroni, P. Gervasio, *A Primer on Mathematical Modelling*, UNITEXT 121,
https://doi.org/10.1007/978-3-030-44541-6_1

1.1 Mathematical Modelling

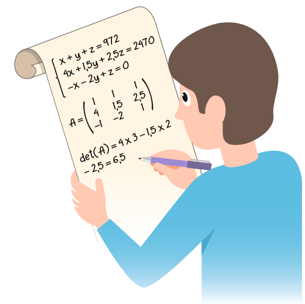
The various aspects of the real world, their interaction and their dynamics can very often be described by mathematical formulas, functions and equations, in other words by *mathematical models*. The purpose of *mathematical modelling* is to translate certain problems arising from the real world into mathematical problems, and provide possible solutions.

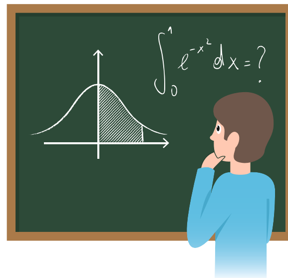
Rather than a subject, it is a process that starts from a real phenomenon or problem of practical interest (from now on called *physical problem*), and aims to:

1. detect the aspects that are most relevant and meaningful;
2. translate them into a mathematical model;
3. compute the solution of the model;
4. verify its soundness, by comparing the solution thus found to the observed (and quantified) phenomenon.

In the study of complex problems very often computing the solution of the model by hand, on paper, is difficult, if not impossible. This difficulty can be attributed to two distinct factors.

Complexity. In certain situations, even if we know the abstract formula for the solution, using it would involve an enormous amount of calculations and take too much time, sometimes even years or decades (as we shall see in Chap. 2, Sect. 2.2).





Unawareness of the solution. In other cases (the vast majority) we cannot even characterise the solution of the mathematical model, meaning we are not able to write a formula to solve the problem explicitly (see Chap. 2, Sect. 2.1).

Numerical modelling and *scientific computing* will help us overcome these obstacles.

1.2 Numerical Modelling and Scientific Computing

Numerical models are examples of formulas and equations, just like mathematical models. What distinguishes them is that the solution to the former, called the *numerical solution*, can be calculated by *numerical methods*, i.e. methods which can be readily translated into *algorithms* and then *programs* executable by a *computer*.

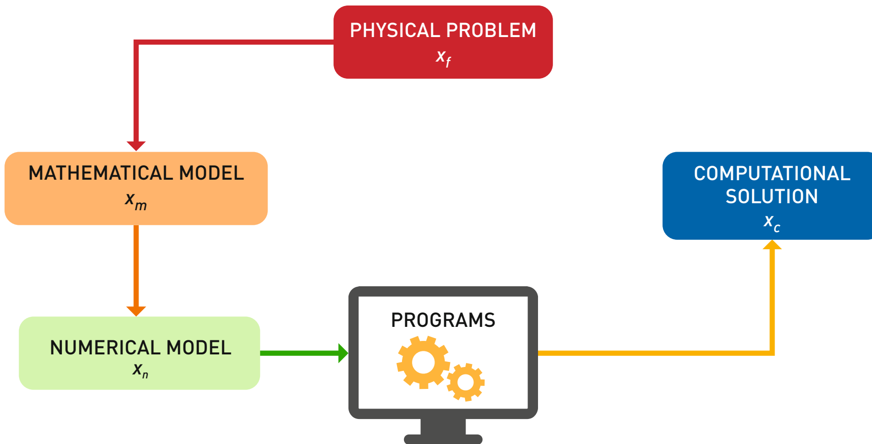


Fig. 1.1 From the physical problem to the computational solution: x_f is the solution to the physical problem, x_m the solution to mathematical model, x_n the solution to the numerical model, x_c the computational solution (produced by the computer)

Therefore numerical models show up in the process, after mathematical models, as Fig. 1.1 shows.

Numerical models represent the bridge between mathematical models and a computing machine, which may be the calculator app on our phones or the most powerful supercomputer in the world, one that is able to perform millions of billions of operations per second.

The final stage of the whole process of modelling a physical problem is the solution generated by the computer, which is called the *computational solution*.

Scientific Computing

Scientific Computing is the subject that takes a mathematical model, cooks up a numerical model, proposes algorithms to determine the computational solution, and then verifies how correct and precise the results of the computer are by comparing them with the real situation.

Here is a more detailed characterisation of mathematical/numerical modelling. It is a process that, starting from a physical problem, aims to

1. detect the aspects that are most relevant and meaningful;
2. translate them into a mathematical model;
3. formulate a numerical model;
4. suggest algorithms for solving the numerical model;
5. translate the algorithms into programs which, once implemented, provide the computational solution;
6. verify the soundness of the computational solution by comparing it with observations and measurements, i.e. with what truly happens with the initial phenomenon.

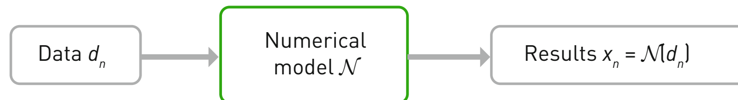
1.3 Data–Model – Results

We may view a mathematical model $\mathcal{M}$ as a tool that acts on the *data* d_m (approximations of the data d_f of the physical problem) and produces the *solution* (or more generally the *results*) x_m. The subscript m reminds us of the word 'mathematical'.

The data d_m are approximations of the physical data d_f, just like x_m is an approximation of the solution x_f of the physical problem we want to solve. We may picture the process by this chart:

The power of the mathematical model $\mathcal{M}$ is that it can act on different data without changing its essence. Clearly the results will depend both on the data and the model we have chosen.

The numerical model $\mathcal{N}$, instead, is a tool acting on an approximation d_n of the data d_m and producing the numerical solution x_n, itself an approximation of the solution x_m. The subscript n refers to *numerical*:



At last, the computational solution generated by the computer, through the program that implements the numerical method, is denoted by x_c. Since, as we shall see later, the computer inevitably introduces errors, *in general the numerical solution and the computational solution are not the same*.

Existence and uniqueness

We are assuming that our mathematical model $\mathcal{M}$ admits one, and only one, solution corresponding to each choice of datum d_m and similarly, that the numerical model has exactly one solution for every choice of datum d_n. This property (the *existence and uniqueness of the solution*) will be a necessary requirement of all mathematical and numerical models we shall meet in the sequel.

1.3.1 Different Physical Problems, the Same Mathematical Model (a Subtle, Yet Crucial, Difference)

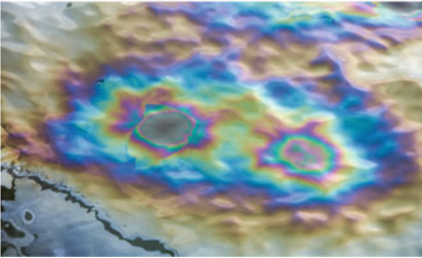
A single mathematical model can be employed for many different problems having common features.

For example, one mathematical model $\mathcal{M}$ that solves several problems in fluid dynamics (air, water, ...) is the so-called system of *Navier-Stokes equations*.[1]

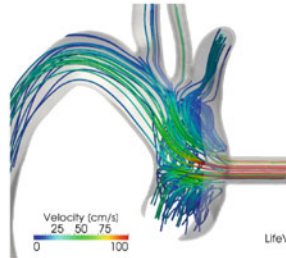
[1] Claude-Louis Navier (1785–1836) was a French engineer and scientist; George Gabriel Stokes (1819–1903) was an Irish mathematician and physicist.

(a) Photo by Ricard Patou from Pixabay, 2018; id 3388188

(b) Photo by Waltteri Paulaharju from Pixabay, 2014; id 1263169

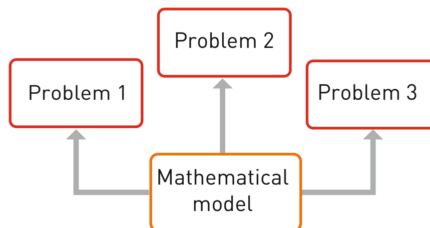


(c) Photo by Dennis Larsen from Pixabay, 2013; id 1438862



(d) Image by CMCS-EPFL, Lausanne (Switzerland) 2017

Fig. 1.2 Different physical problems described by the same mathematical model (the Navier-Stokes equations): trouble waters in a river (**a**); water waves around a rowing hull (**b**); mixing of two fluids with different densities (**c**); streamlines in a bifurcating artery (the computational solution) (**d**)



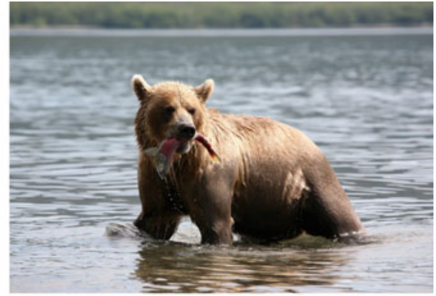
These equations simulate the motion of fluids in canals, rivers, basins (seas or lakes), around floating bodies (a boat, a canoe or a swimmer), or inside the blood vessels of the human body (see Fig. 1.2).

The data d_f of the physical problem are for instance represented by the shape of the riverbed or the lake, the profile of a swimmer or boat, and the features of the fluid under consideration (such as its viscosity and density). The results x_f are the velocity and pressure of the fluid at each point of the geometrical region in which it flows.

Mathematical models are usually grouped according to the type of problems they can tackle. There are mathematical models to study fluid motion as mentioned above, the deformation of solid constructions (say, buildings and bridges subject to the seismic waves of an earthquake), the dynamics of animal populations competing for survival, the evolution of financial markets, the spread of epidemics, and many more phenomena (see Fig. 1.3).

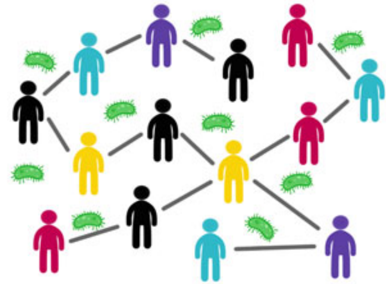
Let us point out that there are even more general mathematical models which are capable of solving problems of very different nature. A single mathematical model

(a) Photo by Angelo Giordano from Pixabay, Onna (Italy) 2009; id 1665878



(b) Photo by Natalia Kollegova from Pixabay, 2009; id 2514853

(c) Photo by Gerd Altmann from Pixabay, 2016; id 1426331



(d) Image by Caterina Saleri, 2020. Reproduced with permission

Fig. 1.3 Types of real problems: the study of deformations of structures subject to earthquakes (**a**); the study of problems in ecology (**b**); the study of the behaviour of financial markets (**c**); the study of the spread of epidemics (**d**).

may for example allow to understand how heat propagates across the walls of a building, what is the electric potential generated by a certain distribution of charge in the surrounding space, or how a soluble substance disperses in a solution: in this latter case a model could describe the concentration of a drug in a watery solution[2] (see Fig. 1.4).

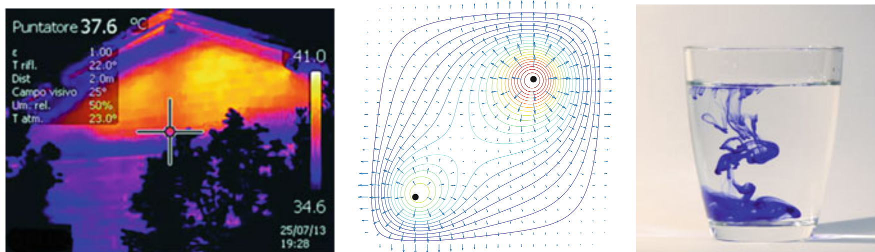
Model

A *model* is a general (and natural) law with major descriptive power.

1.3.2 One Physical Problem, Many Mathematical Models

We claimed above that the solution $x_m = \mathcal{M}(d_m)$ not only depends on the data, but on the chosen model $\mathcal{M}$ as well.

[2]The Poisson equation is $-\Delta u = f$, where f is a given function, u is an unknown function and Δ indicates the sum of the pure second derivatives of u (we shall discuss derivatives in Chap. 6).

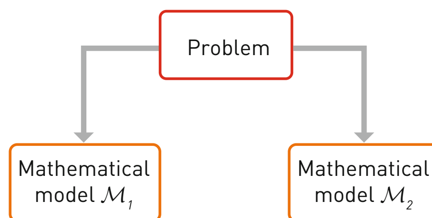


(a) Photo by Marco Danesi, in *Eutopia Urbanscape, the combined redevelopment of social housing*, B. Angi (ed.), LetteraVentidue, Siracusa, ISBN: 9788862421904, 2016. Reproduced with permission

(b)

(c) Photo by Sarah Richter from Pixabay, 2016; id 2427263

Fig. 1.4 The mathematical model given by the Poisson equation, and three physical problems it solves: the thermal insulation of a building and the propagation of heat through the walls (**a**); the electric potential generated by two ideal charges (**b**); the diffusion of a drop of soluble substance in a liquid (**c**)



Let us consider the problem of determining how the polluting particles emitted by a chimney disperse in the atmosphere. We may start by defining a mathematical model $\mathcal{M}_1$ to find out the concentration of a polluting agent by only considering the phenomenon of *diffusion* of a substance (the pollutant) inside another substance (air). This model produces a solution $x_1 = \mathcal{M}_1(d_1)$.

Then we may consider a mathematical model $\mathcal{M}_2$ which, apart from simulating the effects of the diffusion, also considers the *wind* blowing around the chimney. The solution $x_2 = \mathcal{M}_2(d_2)$ produced by the second model differs from solution x_1 when the effect of the wind is significant, and turns out to be more faithful than the solution of the simpler model $\mathcal{M}_1$.

Simplification

We have built two mathematical models to describe the same physical problem, but with different degrees of accuracy. Each time we make a simplification we are actually ignoring one aspect of reality, and thus we are introducing an approximation of the phenomenon. The fewer simplifications we make, the more accurate the solution we compute will be, and the bigger the effort needed to find it.

1.4 Errors, Errors… But No Panic

The Error of the Model

When we pass from a real-life problem to a mathematical model we introduce an error. This happens because reality will not let itself be easily "reduced" to an equation or a collection of equations, irrespective of how sophisticated they are. The price to pay for using a mathematical model is quantified by the discrepancy between the real solution x_f and the (theoretical) solution x_m of the mathematical model (see Fig. 1.5). This difference is the *error of the model*:

$$e_m = x_f - x_m.$$

The Numerical Error

A second type of error, called *numerical error*, measures the difference between the solutions of the mathematical and numerical models:

$$e_n = x_m - x_n.$$

Numerical errors are caused by the approximation of mathematical "operations".

For example, we produce a numerical error when we approximate the area of a *trapezoid*, the region in the plane between the x-axis, two vertical lines and the graph of a function $y = f(x)$ (in blue in Fig. 1.6). Here x_m is the exact value, while x_n is the approximation given by adding the areas of the rectangles (coloured in red in Fig. 1.6). The red rectangles all have basis of the same length $\Delta x > 0$ but differ by height, which in each rectangle equals the value of the function f at the midpoint of its basis.

The numerical error $e_n = x_m - x_n$ measures the difference between the exact area and the approximate area, and depends on the length Δx of the bases. It is intuitively clear that the error will become smaller as Δx becomes small, or equivalently, as the number of rectangles increases.

We also make a numerical error when we set out to determine the speed of a moving car and we approximate the speed v at the instant t using the average speed over a time interval (see Fig. 1.7). The average speed $\bar{v}$ is by definition the ratio of the variation of the position $\Delta s = s_2 - s_1$ over the time interval $\Delta t = t_2 - t_1$ of the

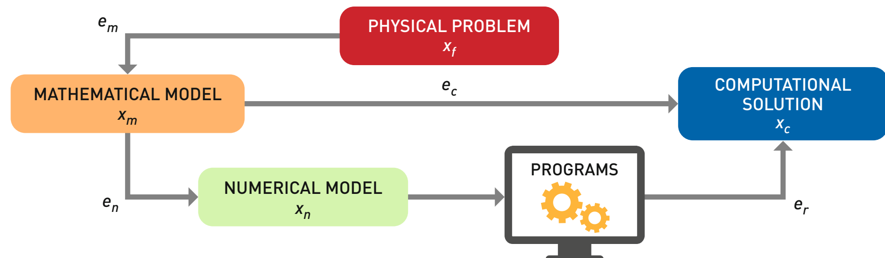


Fig. 1.5 The errors introduced along the path going from the physical problem to the computational solution: the error of the model e_m, the numerical error e_n, the roundoff errors e_r, and the computational errors e_c

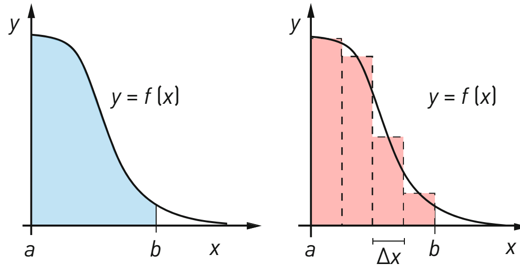


Fig. 1.6 The solution x_m of the mathematical problem is the area of the blue trapezoid, while the solution x_n to the numerical problem is the sum of the areas of the red rectangles. The numerical error $e_n = x_m - x_n$ is the difference between the exact blue area and the approximate area

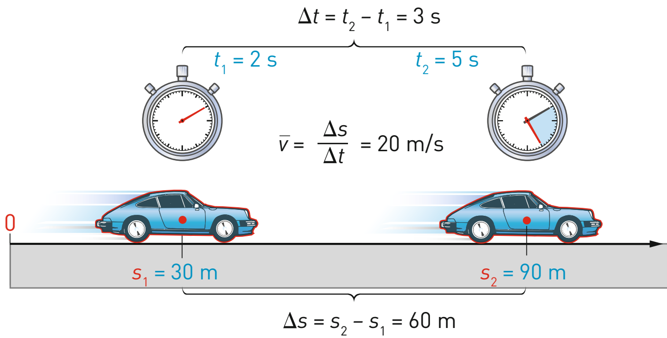


Fig. 1.7 Average speed of an object in motion

displacement; t_1 and t_2 are two instants at which we can measure the position of the car, while the instant t, at which we seek the instant speed, is between t_1 and t_2.

In this case the solution of the mathematical model is $x_m = v$ (the speed at time t), whereas the solution to the numerical model is

$$x_n = \bar{v} = \frac{\Delta s}{\Delta t} = \frac{s_2 - s_1}{t_2 - t_1}.$$

If the speed is not constant, we will make a numerical error

$$e_n = x_m - x_n = v - \bar{v}.$$

The *Tutor* speed-control system on motorways does exactly that: it computes the average speed over a stretch of motorway rather than the instant speed at one point. Two cameras, placed for instance 1 km apart ($\Delta s = 1$ km), detect the instants t_1, t_2 at which a specific vehicle passes by, and transmit the data to a computer that calculates the average speed

$$x_n = \bar{v} = \frac{\Delta s}{\Delta t} = \frac{1 \text{ km}}{t_2 - t_1}$$

over that stretch of road. Obviously this system cannot measure the exact speed at a given instant, and this speed may vary quite a bit, depending on overtaking and braking.

There also are situations where the numerical model coincides with the mathematical model (for example when the mathematical model is a linear system, as we shall see in Chap. 2), $x_n = x_m$, so the numerical error is zero.

Roundoff Errors and Computational Errors

This is not the end of the story. The program used to implement the numerical method on a computer will introduce further *errors* e_r, called *roundoff errors*, due to fact that the algebraic operations performed by the computer are themselves affected by errors.[3]

So we will have:

$$e_r = x_n - x_c.$$

Furthermore, the sum of the numerical and roundoff errors produces *computational errors*:

$$e_c = e_n + e_r = (x_m - x_n) + (x_n - x_c) = x_m - x_c.$$

It is almost by a miracle (so to speak... it is mathematics that enables us to do so) that all these errors can be kept in check. If we follow the rules the computational solution will very much look like the real one[4]

$$x_c \simeq x_f,$$

and only in such case we can say that the mathematical model was helpful.

The Validation of the Process and the Verification of the Numerical Model

The process of comparing the computational solution and the physical solution, that is to say, the control of the error,

$$x_f - x_c = (x_f - x_m) + (x_m - x_c) = e_m + e_c,$$

is also known as *validation of the whole process* consisting in approximating x_f by x_c.

For this process to be validated it must be *verified* first. By *verification of the numerical model* we mean the comparison of the solutions of the mathematical and

[3] Any computer rounds off the input and output of its operations, because all numbers must be stored using a finite number of bits. The resulting arithmetic is called *floating-point arithmetic* to distinguish it from *exact arithmetic*, the one we would do by hand. These notions will be developed further in Sect. 3.2, Chap. 3.

[4] ($a \simeq b$ is read *a is approximately equal to b*).

numerical models

$$x_n \simeq x_m,$$

which amounts to having control on the numerical error e_n. The verification of a numerical model, therefore, is a necessary condition for the validation of the entire process.

What Do Errors Depend On?

The different kinds of errors depend on a number of factors. In a nutshell, for the numerical error we may assume that

$$e_n = e_n(h),$$

in other words that e_n depends on a parameter $h > 0$ that denotes the generic *discretisation parameter* (in the previous examples, h is the width Δx of the intervals used to approximate the area of the trapezoid, or Δt when we approximate the instant speed by the average one).

Similarly, for the roundoff error we may suppose

$$e_r = e_r(\mathtt{u}),$$

so that e_r depends on a parameter $\mathtt{u}$,[5] called *roundoff unit*, which measures how much our computer is accurate in approximating real numbers and in performing algebraic operations.

The error $e_r(\mathtt{u})$ is a measure of how much the calculation depends on the computer itself. This precision is finite, because the machine cannot work using exact arithmetic (as we would, ideally). We shall address numerical errors in Chaps. 2 and 6, and roundoff errors in Chap. 3.

The most complicated error to quantify is the error e_m of the model, for it depends on the physical problem and on how well the adopted mathematical model is able to represent nature faithfully.

Only the final validation, that is, the comparison of the computational solution with the measurements of observable physical quantities (for instance, the temperature predicted by a weather forecast model and the actual temperature detected) will tell us how faithful and reliable the mathematical model we have adopted is. This is the true, ultimate test for a mathematical model.

Approximation

It is inevitable for the computational solution to be imprecise: it will always give an approximation of the exact (and seldomly computable) solution of the mathematical model.

---

[5] What is $\mathtt{u}$ exactly? When we employ Octave (this program will be useful later) we have $\mathtt{u} \simeq 10^{-16}$. For pocket calculators typically $\mathtt{u} \simeq 10^{-9}$.

The problems discussed in this book have smallish dimension, and the corresponding roundoff errors are indeed negligible. For this reason in the next chapters we will always speak of *numerical solutions*, even when we shall actually be considering computational solutions.

We shall hence assume $x_c = x_n$, that is: $e_r = 0$.

1.5 There Is Error and Error

So, making *errors* when we use mathematical models is inevitable. And as a matter of fact, we want this! Should we worry about it? Sure, but just a little. Or rather, we should not worry too much, provided there are mathematical justifications in place when we do so. What really matters is being able to "quantify" these errors, suggest estimates for their size. We know we are generating errors, but we should also be aware of how much we deviate.

As we have seen in the previous section, concerning the area of a trapezoid, the numerical solution x_n is typically obtained in terms of a parameter $h > 0$, called discretisation parameter. We shall emphasise this dependency by writing $x_n(h)$.

The hope is that by taking increasingly smaller positive values of h (closer and closer to zero) the numerical solution $x_n(h)$ will get closer to the solution of the mathematical model x_m (see Fig. 1.8, where $h = \Delta x$).

If this happens, we write

$$x_n(h) \to x_m \qquad \text{as} \qquad h \to 0.$$

We spell out the above as follows: $x_n(h)$ tends to x_m as h tends to 0.[6]

Consequently, as h becomes smaller the numerical error $e_n(h) = x_m - x_n(h)$ will approach zero:

$$e_n(h) \to 0 \qquad \text{as} \qquad h \to 0,$$

read: $e_n(h)$ tends to 0 as h tends to 0.

Convergence

If

$$e_n(h) \to 0 \qquad \text{as} \qquad h \to 0 \tag{1.1}$$

we say that the numerical method is *convergent*, or that it *converges*.

For example, the numerical method that approximates a trapezoid by a sum of rectangles of width $h = \Delta x$ (Fig. 1.8) is a convergent method, because one can prove that the error $e_n(h) = x_m - x_n(h)$ tends to zero as h tends to zero.

[6] In more refined terms: x_m is the limit of $x_n(h)$ as h tends to 0.

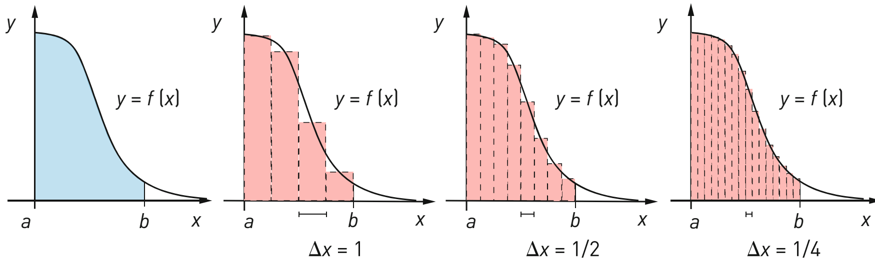


Fig. 1.8 The numerical solution x_n is the sum of the areas of the rectangles, and its value depends on the discretisation parameter $h = \Delta x$. The numerical error $e_n = x_m - x_n$, i.e. the difference between the exact and approximate areas, becomes as small as we want, as $h = \Delta x$ becomes small

In fact, Fig. 1.8 shows that as h decreases, the region obtained by adding the rectangles becomes indistinguishable from the trapezoid.

The proof of this fact requires notions and theorems from infinitesimal calculus.

Convergence of order p

Suppose there exist a positive constant C, independent of h, and a positive integer p (also independent of h) such that the numerical error can be bounded in terms of h as follows

$$\left|e_n(h)\right| \leq Ch^p \quad \text{as} \quad h \to 0. \tag{1.2}$$

Then we say that the *numerical method converges with order* p.

Referring to the approximation of the area of the trapezoid, one can prove that the following error estimate holds:

$$\left|e_n(h)\right| \leq Ch^2 \qquad \text{as } \; h \to 0,$$

for a suitable positive constant C, and we say that the method converges with order 2 with respect to h.[7] We shall have the opportunity to understand this method better in Chap. 2.

At times we may also replace the symbol $\leq$ by $\approx$ (read "behaves like") in the error estimate, and thus write

$$\left|e_n(h)\right| \approx h^p \quad \text{as} \quad h \to 0.$$

This amounts to say there exist positive constants C and c, independent of h, for which

$$ch^p \leq \left|e_n(h)\right| \leq Ch^p \quad \text{as} \quad h \to 0.$$

In other words, the absolute value of the error behaves exactly like h^p when $h \to 0$.

[7] Halving h, for example, i.e. taking twice as many intervals, we reduce the error by a factor of 4.

Error Estimators

A careful inspection tells that errors are not computable quantities, for they depend on a constant C that in turn depends on the unknown of the problem itself, which we obviously do not know.

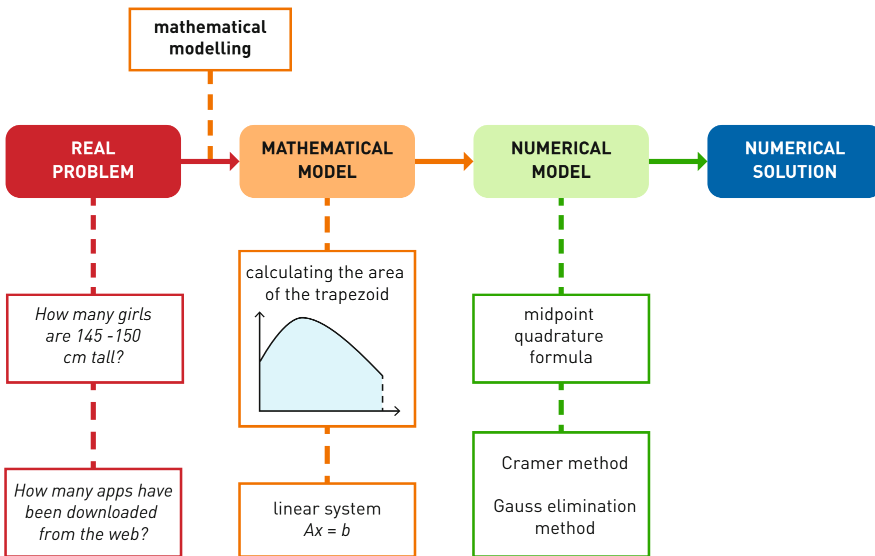
So sometimes it becomes necessary to introduce *error estimators*: computable quantities that give a good indication of the error made. We will meet some instances in the subsequent chapters.

1.6 What We Have Learnt

We have learnt:

1. what it means to make a *mathematical model*;
2. that mathematical models often require approximations by *numerical models*, which are implementable on *computers*;
3. that numerical models, by nature, provide *error-affected* solutions (whence numerical solutions may be different, in principle, from the incomputable solutions of the mathematical model);
4. that, despite everything, errors can be *predicted and measured*.

2 Getting Started



INGREDIENTS	Section 2.1: definition of exponential function. Section 2.2: 2×2 and 3×3 linear systems.
WHAT WE LEARN	Section 2.1: trapezoid, numerical calculation of areas. Section 2.2: elementary notions about vectors and matrices, linear systems in vector form, Cramer method, Gauss elimination method (GEM) and corresponding algorithms, computational costs.
PREREQUISITES	Chapter 1.

Let us begin with two simple problems (a girl-fashion company, and an app developer). They will serve as an excuse to start formulating mathematical models and discuss interesting algorithms, in view of finding solutions.

© Springer Nature Switzerland AG 2020

A. Quarteroni, P. Gervasio, *A Primer on Mathematical Modelling*, UNITEXT 121,
https://doi.org/10.1007/978-3-030-44541-6_2

During this study we shall make a detour on how to compute the area of a trapezoidal region and how to solve linear systems efficiently.

2.1 When the Solution to the Mathematical Model Is Not Known

Even if we might be able to characterise the solution of the model, rather often there is no formula to compute said solution explicitly, and we can only hope to approximate it using numerical methods. Here is an example.

Problem "Heights"

A company that sells clothing for 8-to-14-year-old girls wants to optimise its manufacturing process. It is therefore interested in estimating how many girls, in a sample group of 1000, are between 145 cm and 150 cm tall. This is the information at hand:

- the average height x_m among the 1000 girls is 145 cm;
- the *standard deviation* σ equals 3 cm (σ indicates how much, on average, the height of each girl is above or below the 145 cm average);
- the function describing the distribution of height values around the average value is[1]

$$f(x) = \frac{1000}{\sigma\sqrt{2\pi}} e^{-\frac{(x-x_m)^2}{2\sigma^2}} \qquad \text{where } x_m = 145 \text{ and } \sigma = 3.$$

Statistics and infinitesimal calculus provide us with a formula to express the number of girls, among the 1000, whose height lies in the range 145–150 cm. On the graph of the function $f(x)$, the required value can be estimated by the area of the blue trapezoid T in Fig. 2.1. This is the region enclosed by the x-axis, the vertical lines $x = x_1 = 145$, $x = x_2 = 150$ and the graph of $y = f(x)$ (the latter is the curve whose points correspond to the pairs $\big(x, f(x)\big)$).

The mathematical model for problem "Heights"

The *mathematical model* of the problem therefore consists in determining the area of the trapezoid T bounded by the horizontal axis, by the vertical lines $x = x_1 = 145$, $x = x_2 = 150$ and by the graph of $y = f(x)$. The model also encodes the description of the data using the function $y = f(x)$ given by the problem, a function which in turn depends on the average $x_m = 145$ and the standard deviation $\sigma = 3$.

[1] The function $f(x)$ is represented in Fig. 2.1. If we adopt the terminology used in statistics, $f(x)$ is the *density function* associated with the distribution. The particular distribution of our problem is known as *normal* (or *Gaussian*) distribution. In statistics one would say that the heights are *distributed normally* with an average value of 145 and standard deviation 3.

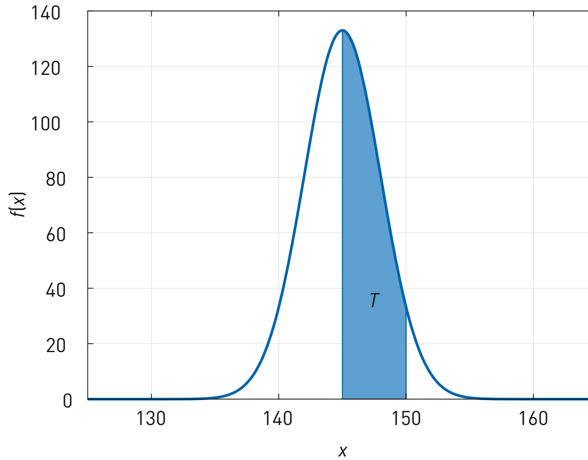


Fig. 2.1 The graph of $f(x)$ in problem *Heights*. The area of the trapezoid T gives the approximate number of girls with height between $x_1 = 145$ cm and $x_2 = 150$ cm

In this way we detect:

- the *data of the problem*: the average height $x_m = 145$ cm, the standard deviation $\sigma = 3$, the function $y = f(x)$ describing the height distribution;
- the *mathematical model*: computing the area of the trapezoid T bounded by the graph of $y = f(x)$;
- the *solution*: the value of the area T in Fig. 2.1.

How Do We Compute the Area of the Trapezoid *T*?

Infinitesimal calculus allows us to express this area using integrals. The area of our trapezoid T is in fact given by the integral:

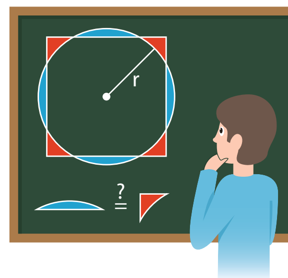
$$\text{area}(T) = \int_{x_1}^{x_2} f(x)dx = \int_{145}^{150} \frac{1000}{3\sqrt{2\pi}} e^{-\frac{(x-145)^2}{18}} dx.$$

Although we have eventually formulated the problem in mathematical terms, this does not help us solve it. In contrast to other integrals, in fact, this particular one cannot be computed by elementary techniques.

Quadrature formulas

Unfortunately there is no formula that gives the exact solution to this problem. We can, however, compute an approximate value by *numerical methods* known as *quadrature formulas*.

The name *quadrature formula* originates in a question going back to Ancient Greece, namely the *quadrature of the circle*. That problem asked to construct a square with the same area as a given circle using only straight-edge and compass.



Let us now discuss a quadrature formula to approximate the area of the trapezoid T of Fig. 2.1.

2.1.1 Quadrature Formulas, or How to Compute Areas Approximately

We shall consider a more general context than that of problem *Heights*. Let us look at a function $y = f(x)$ defined on a closed and bounded interval $[a, b]$ that only assumes positive values, like the one in Fig. 2.2. The goal is to compute the area of the trapezoid T enclosed by the x-axis, the lines $x = a$, $x = b$ and the graph of $y = f(x)$.

Quadrature nodes

What is special about a quadrature formula is that it approximates a trapezoidal area by using only the values of f at suitable points x, called *quadrature nodes*, in the interval $[a, b]$.

'Divide et impera': Let Us Subdivide the Interval

Choose an integer $M > 0$ and define $h = \dfrac{b-a}{M}$. Then select evenly distributed points (see Fig. 2.3)

$$x_0 = a,\ x_1 = x_0 + h,\ x_2 = x_1 + h,\ \ldots,\ x_{M-1} = x_{M-2} + h,\ x_M = b.$$

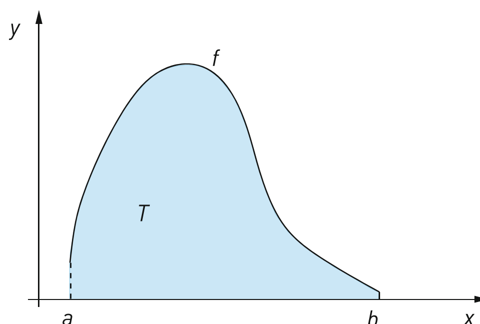


Fig. 2.2 The function f defined over the interval $[a, b]$ and the blue trapezoid T of which we seek the area

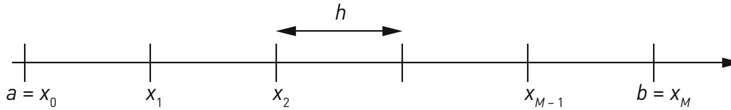


Fig. 2.3 Evenly spaced points inducing the partition of the interval $[a, b]$

Now divide $[a, b]$ into M smaller intervals as follows: I_1 is the interval between x_0 and x_1, so $I_1 = [x_0, x_1]$; then take $I_2 = [x_1, x_2]$ and in general, for every $k = 1, 2, \ldots, M$, denote by $I_k = [x_{k-1}, x_k]$ the generic subinterval.[2]

The goal is to *approximate* (read: substitute), over each subinterval, the initial function f with a "simpler" function; then, instead of the area of T, to compute the area of the trapezoid of the new function.

Up and Down, Step by Step

Starting from the partition of $[a, b]$ into M intervals of the same length h, we introduce the *step* function f_{mp} depicted in Fig. 2.4.

A function like f_{mp} is called *step function* because it has the shape of a staircase, albeit an unusual one: the steps have different heights and some go up, others go down.

Let us explain how to define f_{mp} starting from f and the partition.

Call $\overline{x}_1, \overline{x}_2, \ldots, \overline{x}_M$ the midpoints of the intervals $I_1, I_2, \ldots, I_M$:

$$\overline{x}_k = \frac{x_{k-1} + x_k}{2} \qquad \text{for } k = 1, \ldots, M.$$

On each interval I_k we set f_{mp} to be the constant function that assumes the value of f at the midpoint $\overline{x}_k$ (Fig. 2.4):

$$f_{mp}(x) = f(\overline{x}_k) \qquad \text{for every } x \in I_k, \text{ with } k = 1, 2, \ldots, M.$$

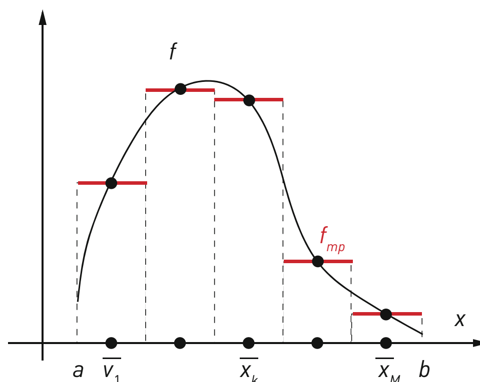


Fig. 2.4 The given function f (black) and the "step" function f_{mp} (red) over $M = 5$ intervals

[2] When we write $k = 1, 2, \ldots, M$ we mean that the preceding formula is valid "for every k from 1 to M": in other words k is a natural number whose range are all integers between 1 and M.

Approximation

The step function f_{mp} is therefore an approximation of f.

The subscript mp stands for *midpoint* since, in order to define f_{mp}, we have used the values of f at the midpoints of the subintervals.

Exercises

Solve Exercise 2.1, p. 203.

As shown in Fig. 2.5, f_{mp} defines a trapezoid T_{mp} as well: the union of the rectangles with bases the I_k.

Quadrature Formula at Midpoints, or by Rectangles

Just as f_{mp} is an approximation of f, the area of the trapezoid T_{mp} associated with f_{mp}, too, is an approximation of the area of T (see Fig. 2.5):

$$area(T) \simeq area(T_{mp}),$$

where $\simeq$ should be read as "approximately equal to", and expresses an approximated value.

At this point computing $area(T_{mp})$ is indeed very easy, for it amounts to adding the areas of the rectangles over the intervals I_k, which have base h and as height the value of f at the midpoints $\overline{x}_k$, that is[3]

$$\begin{aligned} area(T_{mp}) &= hf_{mp}(\overline{x}_1) + hf_{mp}(\overline{x}_2) + \ldots + hf_{mp}(\overline{x}_M) \\ &= hf(\overline{x}_1) + hf(\overline{x}_2) + \ldots + hf(\overline{x}_M) \\ &= h(f(\overline{x}_1) + f(\overline{x}_2) + \ldots + f(\overline{x}_M)) \\ &= h\sum_{k=1}^{M} f(\overline{x}_k). \end{aligned}$$

Numerical solution

We conclude that the value

$$area(T_{mp}) = h\sum_{k=1}^{M} f(\overline{x}_k) \tag{2.1}$$

gives an approximation of the exact area of T. It is a *numerical solution* and its value depends on the number M of intervals considered.

[3] The symbol $\sum_{k=1}^{M}$ is read *sum as k goes from 1 to M*.

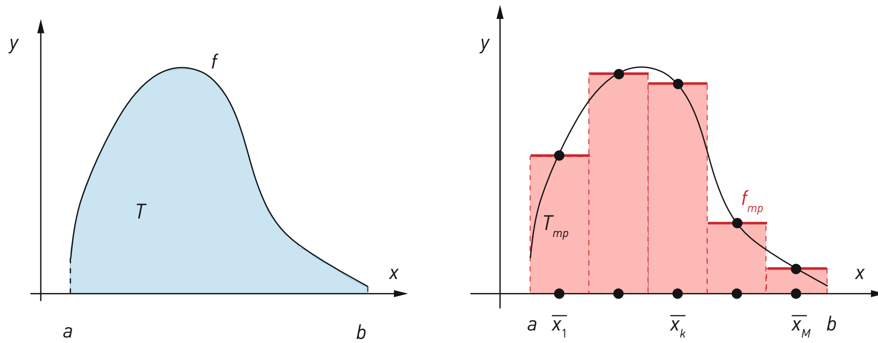


Fig. 2.5 The trapezoid T (blue) associated with f, the trapezoid T_{mp} (red) associated with f_{mp}

The points $\overline{x}_k$ at which we evaluate f in order to compute $area(T_{mp})$ are called the *quadrature nodes* of the midpoint formula.

Exercises

Solve Exercise 2.2, p. 203.

We see from Fig. 2.6 that the trapezoid T_{mp} contains areas (in red) that lie outside of T, and at the same time T_{mp} leaves out certain parts of T (those in blue). The values of these areas represent *errors*: in red rounded-up errors, in blue rounded-down errors.

If the sum of the red areas equals the sum of the blue ones, then $area(T) = area(T_{mp})$. This will definitely happen when $f(x)$ is a polynomial function of degree 1 (a straight line). In any other case $area(T_{mp})$ will just be an approximate value of $area(T)$.

In the general situation where f is not a degree-one polynomial, it is natural to believe that by increasing the number of intervals, the value of $area(T_{mp})$ will approximate better the exact value $area(T)$. It is also natural to ask ourselves how many intervals M should one use for this.

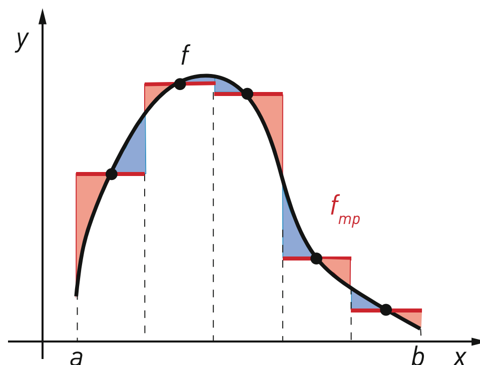


Fig. 2.6 Overestimates (in red) and underestimates (in blue) in the midpoint formula

Unfortunately there is no clear-cut answer, one that is valid in general for any type of problem.

Accuracy vs number of operations

The optimal number M of intervals for approximating $area(T)$ by $area(T_{mp})$ depends on the features of the function $f(x)$, on the accuracy we require to approximate the exact solution, and on how many *elementary operations* (sums, subtractions, multiplications and divisions) we allow the computer to perform. In this respect, we should not forget the rule of thumb whereby *the higher the number of elementary operations, the longer the computer will take to answer.*

Are We There Yet?...Not Quite

So now we have a formula to approximate the solution to problem *Heights*, yet we are still not completely satisfied...

It is not enough to have a method to compute the numerical solution. What we really want (and need) to understand, before we go on, is how this numerical solution actually approximates the exact solution. Put otherwise, we wish to quantify the numerical error we are making.

2.1.2 The Errors in the Quadrature Formula

We saw above that when $f(x)$ is not a polynomial of degree one, the value $area(T_{mp})$ is just an approximation of the exact value $area(T)$, and that it depends on the length h of the base intervals.

Resuming the terminology introduced in Chap. 1, we have:

- $x_m = area(T)$ is the mathematical solution;
- h is the discretisation parameter;
- $x_n(h) = area(T_{mp})$ is the numerical solution that approximates the mathematical solution, and it depends on h;
- $e_n(h) = x_m - x_n(h)$ is the *numerical error*, also depending on h.

Questions...

1. Suppose we take a larger and larger number M of intervals (mathematically, $M \to \infty$, read 'M goes to infinity'), and hence a length h that is positive but increasingly smaller (written $h \to 0$). Does the absolute value of the error $e_n(h)$ decrease? In mathematical jargon: *is the numerical method convergent*, according to definition (1.1) seen in the first chapter?
2. If the answer is yes, what is the *order of convergence* of the method with respect to h, according to definition (1.2)?

...and Answers

It is possible to address these questions because of some theorems of infinitesimal calculus, and here is what happens.

Error for the midpoint formula

There exists a positive constant C, depending on f and $[a, b]$, but not on h, such that

$$|e_n(h)| = \left|area(T) - area(T_{mp})\right| \leq Ch^2 \qquad \text{as } h \to 0. \tag{2.2}$$

In other words, the error of the midpoint formula tends to zero *quadratically in h* when h tends to 0. Therefore the midpoint quadrature formula is convergent, and *accurate with order two with respect to h*.

Inequality (2.2) guarantees that the midpoint method converges and that the absolute value of the error decreases like the square of h, as h itself becomes smaller. But it says much more.

Error estimate

Inequality (2.2) allows us to *estimate quantitatively the numerical error* $e_n(h)$ we make when approximating $area(T)$ with the value $area(T_{mp})$ (given by the midpoint formula, with given h), *even if we do not know the exact value of the area of the trapezoid* T.

The quantity Ch^2 is an *estimator* of the error made.

The constant C depends solely on the data of the problem and not on the solution, but to compute its value we need to use derivatives.[4]

2.1.3 The Numerical Solution of Problem "Heights"

Finally we have all the tools necessary to solve problem *Heights*: we know a numerical method for approximating the solution and we can quantify the error even without knowing the exact solution.

Let us make a list of what the numerical problem involves:

- the *data of the numerical problem*: the function $f(x) = \dfrac{1000}{3\sqrt{2\pi}} e^{-\frac{(x-145)^2}{18}}$, the interval $[a, b] = [145, 150]$, the number M of subintervals;
- the *numerical model*: the midpoint quadrature formula (2.1);
- the *numerical solution*: $area(T_{mp})$.

Since we do not know how many intervals M we should take, let us begin with $M = 2$, divide $[a, b]$ in two subintervals of length $h = \dfrac{b-a}{2}$ and apply formula (2.1), to obtain $area(T_{mp}) = 457.0$.

[4]If the reader knows about derivatives, then $C = \frac{b-a}{24}\max_{x\in[a,b]}|f''(x)|$, provided f is twice differentiable.

Table 2.1 The values $area(T_{mp})$ computed using various choices of M (hence of h)

M	h	$area(T_{mp})$
2	2.5	457.0
4	1.25	453.4
8	0.625	452.5
16	0.3125	452.3
32	0.15625	452.2
64	0.078125	452.2

Now take $M = 4$, then $M = 8$ and so on up to $M = 64$, and each time compute $area(T_{mp})$. The outcome values get closer (converge) to 452.2, as Table 2.1 shows.

Although the answer should be a natural number (the number of girls with height in the 145–150 cm range), we wrote down the numerical solutions with one decimal digit in order to better understand how the values $area(T_{mp})$ vary as M varies.

In conclusion, we accept the value 452 as a satisfactory approximation of the correct value, and we say that the number of girls between 145 and 150 cm tall is 452, over a total of 1000.

Can We Determine How Many Intervals to Use Without Wasting Time?

The answer is yes, and now we set out to show how.

Already with $M = 16$ the midpoint quadrature formula gives an accurate result, pretty close to the one obtained with $M = 64$. But how do we know when to accept a numerical solution as valid? And when, instead, should we seek a more accurate one (with larger M), which at the same time will be computationally more costly?

To understand whether the numerical solution corresponding to a certain M is accurate enough, we may use estimate (2.2). It guarantees that the value $area(T_{mp})$ computed using a specific h differs from $area(T)$ at most by Ch^2, which we must determine.

It is not difficult to find $h = \dfrac{b-a}{M}$. As for C, we will just provide the right value $C \simeq 3.0783$, which is found by applying the formula $C = \dfrac{b-a}{24} \max_{x\in[a,b]} |f''(x)|$.

Table 2.2 shows the error estimator Ch^2 corresponding to the values of h used in Table 2.1.

Table 2.2 Error estimator for problem *Heights*

M	h	Ch^2
2	2.5	19.2391
4	1.25	4.8098
8	0.625	1.2024
16	0.3125	0.3006
32	0.15625	0.0752
64	0.078125	0.0188

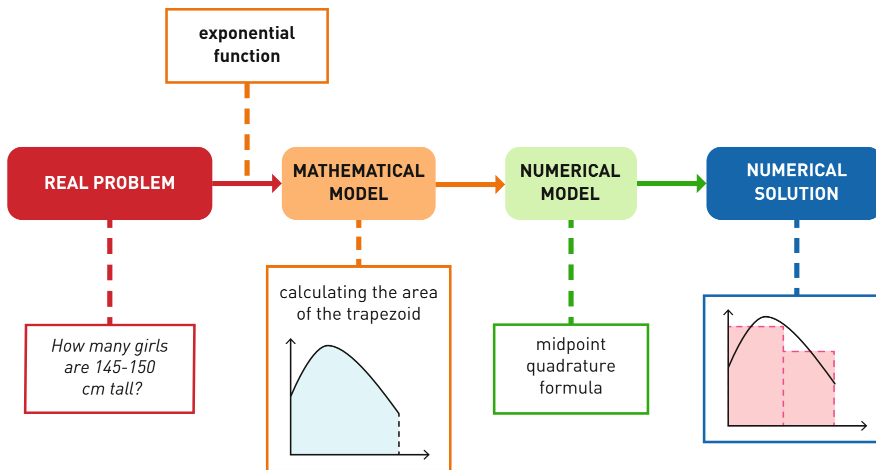


Fig. 2.7 The conceptual map of Sect. 2.1

By choosing $M = 2$ to approximate $area(T)$ we make at most an error of 19.2391. (The error could be much smaller, but this cannot be known in advance.)

By increasing M the errors become smaller, and for $M = 32$, the error is at most 0.0752.

Since we are interested in integer values for the solution ($area(T)$ estimates the number of girls between 145 and 150 cm), an estimate of 0.0752 warrants a highly accurate numerical solution and there is no need for further calculations involving larger values of M. More generally, an error smaller than 0.1 would not change the (integer) value of the solution, since the latter is rounded.

On the other hand, when $M = 16$ the error estimate is 0.3006, so we cannot precisely give an (integer) solution because the margin of error is one unit. In fact for $M = 16$ the area is 452.3, and a 0.3006 error permits to round the number either to 452 (below) or 453 (above).

Therefore $M = 32$ intervals are enough to guarantee the maximum accuracy at the lowest computational cost.

The flowchart in Fig. 2.7 summarises the path that led us to the solution of the problem.

2.2 Vectors and Matrices to Handle Complexity

As we saw in Chap. 1, sometimes, even if we do know the formula to compute the solution to our mathematical model, the actual computation could require too much time. This is the case of a system of linear equations of medium size. In order to understand the cost of solving (in terms of number of operations and computer execution time) we shall start from a simple system of 2 equations in 2 variables. Later we will increase the order of the system, and thus its complexity.

Problem "Two apps"

David is a developer of smartphone apps. In the last month he made €1,200 by selling two apps, called *Sun* and *Moon*. Knowing that he earns €4 for every download of *Sun* and €1.50 for each download of *Moon*, and that there have been 600 downloads altogether, David wants to know the exact number of downloads of each of his apps.

This is the real-world problem. As a matter of fact it is extremely simple, but we will take it as a starting point to tune up.

To find the mathematical model let us denote by x the unknown number of downloads of *Sun* and by y the number of downloads of *Moon*, and then let us translate into equations the information we have: one equation will express the relationship between the single gain and the total sum earned, and another equation will encode the fact that the total number of downloads is the sum of the single downloads.

The mathematical problem reads as follows: find x and y such that

$$\begin{cases} 4x + 1.5y = 1200 \\ x + y = 600. \end{cases} \tag{2.3}$$

It is a system of 2 linear equations in 2 variables. We shall say, for short, a 2×2 linear system, or a system of order 2.

How to solve 2×2 linear systems is typically taught in the first years of high school, for instance by *substitution*. This boils down to the following operations:

1. isolate y in the second equation:
$$y = 600 - x;$$
2. substitute this y in the first equation:
$$4x + 1.5(600 - x) = 1200;$$
3. solve the above equation: $x = \dfrac{300}{2.5} = 120$;
4. substitute this value x in the equation of step 1 to get $y = 480$.

So, first of all we can tell David there have been 120 downloads of *Sun* and 480 of *Moon*. Then we note that this simple method for 2×2 systems does not work automatically for higher-order systems.[5]

[5]The *order* of a linear system of n equations and n variables is the number n of equations (or of variables).

For this reason we recall here two more methods for solving linear systems, which address the issue: the *Cramer method*,[6] and the *Gauss elimination method*[7] that subsumes the substitution method.

Before we proceed let us first rewrite the linear system in a more concise form by using new mathematical objects: vectors and matrices.

2.2.1 Vectors and Matrices

Vectors: Orderly Queues

We begin with the right-hand-side terms of system (2.3) on page 26, i.e. 1200 and 600, and write them one on top of the other as in a box, in the same order they appear in the system (see Fig. 2.8). We are writing a *column vector*

$$\mathbf{b} = \begin{bmatrix} 1200 \\ 600 \end{bmatrix}$$

called the *right-hand-side vector* of the linear system. Similarly, let us gather x and y in one column vector

$$\mathbf{x} = \begin{bmatrix} x \\ y \end{bmatrix}$$

called the *solution vector* of the linear system.

The numbers inside a vector are called its *components*.

In later sections we shall work with vectors having an arbitrary number n of components. Think of a vector $\mathbf{c}$ of 100 components... Do we have 100 letters to

Fig. 2.8 A famous queue... (Photo by Iain Stewart Macmillan, Abbey Road cover, 1969)

[6] Gabriel Cramer (1704–1752), Swiss mathematician.

[7] Johann Friedrich Carl Gauss (1777–1855), German mathematician, astronomer and physicist, one of the greatest mathematicians ever.

name each one of them? Clearly not! But since numbers abound – actually there are infinitely many numbers, one way out is to label the components by c_1, c_2, $c_3, \ldots, c_{100}$, by attaching a subscript to the generic letter c that denotes the vector. A boldface letter $\mathbf{c}$ will indicate the entire vector, while c_k will be the generic component, with k being an integer between 1 and $n = 100$. Thus we will write a generic column vector as follows:

$$\mathbf{c} = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix}.$$

The number n of components of the vector is the *dimension of the vector*.

We will also need to distinguish column vectors, like the above $\mathbf{b}$, $\mathbf{x}$ and $\mathbf{c}$, from *row vectors* such as

$$\mathbf{d} = [d_1, d_2, \ \ldots, d_n].$$

Matrices, or Close-Order Setting?

Consider now the coefficients of the variables x, y inside system (2.3) above, and deploy them as in a close-order setting (see Fig. 2.9), in (horizontal) rows and (vertical) columns inside a box, called a *matrix*:

$$A = \begin{bmatrix} 4 & 1.5 \\ 1 & 1 \end{bmatrix}.$$

The coefficients have been written in a very specific order in the matrix: the first row (from the top) displays the coefficients of the variables in the first equation of

Fig. 2.9 This close-order setting has 3 rows and 6 windows per row (6 columns). (Photo by Albrecht Fietz from Pixabay, Landsberg am Lech (Germany) 2016; id 3240939)

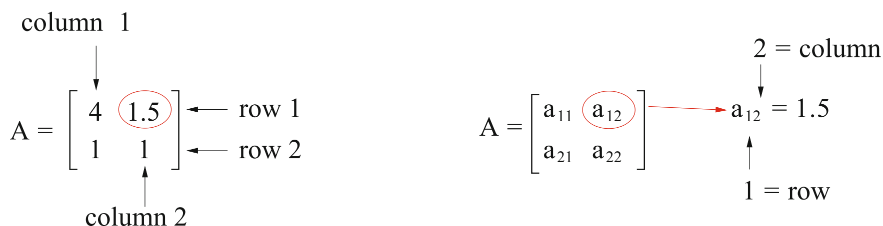
system (2.3), while on the second row we have put the coefficients of the second equation.

What is more, there is a pre-established ordering inside each row: in the first (left-most) column we have written the coefficients of the first variable (x) of the solution vector, in the second column the coefficients of the second variable (y).

The matrix A associated with the linear system has two *rows*, exactly as the number of equations in the system, and two *columns*, equalling the number of variables.

Every entry a_{ij} of a matrix is uniquely determined by two indices, the row and column to which it belongs.

The rows of a matrix are ordered from top to bottom, and the columns from left to right. Since the number 1.5 appears in the first row and second column, it is the element a_{12} of the matrix A.



Analogously, the number 1 in row two and column two is the element a_{22} in A; the number 1 in row two and column one is a_{21} and so forth. More generally, the generic matrix element lying at the crossing of row i (the first index) and column j (the second index) is the element a_{ij}, and a generic matrix A with n rows and m columns can be represented by the tabular array

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{bmatrix}. \tag{2.4}$$

Written more compactly, $A = (a_{ij})$. When A has the same number of rows and columns ($n = m$), we say that A is a *square matrix of order* n.

The elements a_{ii} (that is, a_{ij} when $i = j$) in a square matrix A are called *diagonal elements* because they appear along the ideal diagonal going from the top left corner to the bottom right corner of the square. All other elements a_{ij}, where $i \neq j$, are called *off-diagonal elements*. The *main diagonal* of a square matrix is the set of diagonal elements a_{11}, a_{22}, $\dots$, a_{nn}.

The Product of a Matrix by a Column Vector

Let us know explain what the writing

$$A\mathbf{x} = \begin{bmatrix} 4 & 1.5 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

means. $A\mathbf{x}$ is the product between the matrix A, 2×2 in our case, and the column vector $\mathbf{x}$, which has 2 components. The result of the multiplication is a column vector, still of dimension 2, whose components are found as follows:

1 a. multiply each element of the first row of A by the element in $\mathbf{x}$ in the same position, i.e. $4 \cdot x$ (the first element of the first row of A times the first element in the vector $\mathbf{x}$), and $1.5 \cdot y$ (the second element of the first row of A times the second element of $\mathbf{x}$);
1 b. add up these products, and the result $4x + 1.5y$ is the first component of the final vector;
2 a. multiply each element of the second row of A by the element of $\mathbf{x}$ in the same position, that is $1 \cdot x$ (the first element in row two times the first element in $\mathbf{x}$) and $1 \cdot y$ (the second element in row two times the second element of $\mathbf{x}$);
2 b. add them up, $x + y$, and write this as second element of the resulting vector.

All-in-all

$$A\mathbf{x} = \begin{bmatrix} 4x + 1.5y \\ x + y \end{bmatrix}.$$

The resulting vector $A\mathbf{x}$ contains precisely the terms on the left of the two equations in the system.

Equality of Vectors

At this point we make sense of an equality $A\mathbf{x} = \mathbf{b}$ between vectors.

Recalling how $A\mathbf{x}$ and $\mathbf{b}$ were defined, the equality $A\mathbf{x} = \mathbf{b}$ spells out as

$$\begin{bmatrix} 4x + 1.5y \\ x + y \end{bmatrix} = \begin{bmatrix} 1200 \\ 600 \end{bmatrix},$$

which means that the elements of the two vectors in the same position must coincide. Therefore

$$4x + 1.5y = 1200$$
$$x + y = 600$$

which brings us back to the initial system (2.3).

2.2.2 The 2 × 2 Linear System in Matrix Form

The system associated with problem *Two apps* may be rewritten in an equivalent *matrix form* as follows.

The mathematical model for problem "Two apps"

Given the matrix $A = \begin{bmatrix} 4 & 1.5 \\ 1 & 1 \end{bmatrix}$ and the right-hand-side column vector $\mathbf{b} = \begin{bmatrix} 1200 \\ 600 \end{bmatrix}$, find the vector $\mathbf{x} = \begin{bmatrix} x \\ y \end{bmatrix}$ satisfying the equation

$$A\mathbf{x} = \mathbf{b}.$$

The system in matrix form $A\mathbf{x} = \mathbf{b}$ represents our *mathematical model* for solving the problem.

Let us point out:

- the *data of the problem*: the matrix A and the vector $\mathbf{b}$;
- the *mathematical model*: the system in matrix form $A\mathbf{x} = \mathbf{b}$, equivalent to system (2.3) on page 26;
- the *solution*: the vector $\mathbf{x}$.

2.2.3 The Cramer Method for 2 × 2 Systems

The mathematical notion underpinning the Cramer method is the *determinant* of a matrix.

The Determinant of a 2 × 2 Matrix

Consider a 2×2 matrix

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}.$$

Its *determinant*, written

$$\det(A) \text{ or } \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix},$$

is the real number

$$\det(A) = a_{11} \cdot a_{22} - a_{21} \cdot a_{12}. \tag{2.5}$$

Putting Cramer to Work

The Cramer method requires the following steps.

1. Compute $\det(A)$ using (2.5):

$$\det(A) = a_{11} \cdot a_{22} - a_{21} \cdot a_{12} = 4 \cdot 1 - 1 \cdot 1.5 = 2.5.$$

2. Define the matrix

$$\hat{A}_1 = \begin{bmatrix} 1200 & 1.5 \\ 600 & 1 \end{bmatrix}$$

(obtained from A by replacing the first column with the column vector $\mathbf{b}$) and compute its determinant using (2.5):

$$\det(\hat{A}_1) = 1200 \cdot 1 - 600 \cdot 1.5 = 300.$$

3. Define the matrix

$$\hat{A}_2 = \begin{bmatrix} 4 & 1200 \\ 1 & 600 \end{bmatrix}$$

(obtained from A by replacing the second column with the column vector $\mathbf{b}$) and compute its determinant with (2.5):

$$\det(\hat{A}_2) = 4 \cdot 600 - 1 \cdot 1200 = 1200.$$

4. Compute the solutions of system (2.3):

$$x = \frac{\det(\hat{A}_1)}{\det(A)} = \frac{300}{2.5} = 120,$$
$$y = \frac{\det(\hat{A}_2)}{\det(A)} = \frac{1200}{2.5} = 480.$$

Clearly the solution found by the Cramer method coincides with the one obtained by substitution, the difference only being in the algorithms employed.

2.2.4 The Gauss Elimination Method (GEM) for a 2 × 2 System

Gauss elimination consists of two parts. The first part is the *reduction*, whereby the given system is transformed into an equivalent system (with the same solution as the given one) where certain variables have been eliminated (in the 2×2 case we will only need to eliminate one unknown from the second equation). The second part is the *back-substitution*, where starting from the last equation with one unknown we substitute back, successively, to recover all other variables one by one.

For the sake of simplicity we shall describe GEM on system (2.3), page 26. But let us remind that the same operations can be performed on the matrix form $A\mathbf{x} = \mathbf{b}$ by manipulating the matrix A and the vector $\mathbf{b}$.

1. *Reduction of the system.*
   We want to eliminate x from the second equation of system (2.3) on page 26. To do so we subtract the first equation, multiplied by $\frac{1}{4}$, from the second equation, so to obtain a new equation without x (we say that x has been eliminated):

$$\begin{array}{rrl} \text{(second equation):} & x \;+\; y & = 600 \\ -\frac{1}{4}\cdot\text{(first equation):} & -\frac{1}{4}(4x \;+ \frac{3}{2}y & = 1200) \\ \hline & \frac{5}{8}y & = 300 \end{array}$$

   Then we substitute the second equation of system (2.3) with this new one, giving the new system

$$\begin{cases} 4x + \frac{3}{2}y = 1200 \\ \quad\;\; \frac{5}{8}y = 300, \end{cases} \tag{2.6}$$

   which is equivalent to the initial one (it has the same solutions).

   The factor $\frac{1}{4}$, used to eliminate x, is called *multiplier*, and coincides with the ratio of the coefficients of x in the second and first equations.

2. *Back-substitution of variables.*
   a. Compute y from the second equation of system (2.6): $y = 480$;
   b. substitute this y in the first equation of system (2.6) to get $x = 120$.

In this case, too, the solution found by GEM coincides with the previous ones.

2.2.5 From 2 to 3

Let us increase the complexity of the problem.

Problem "Three apps"

David is very pleased with the prior revenue so he decides to develop another app, called *Earth*, and sell it online for €2.5. One month later he has earned €2,470 from the three apps. He knows there have been a total of 972 downloads altogether, and that the downloads of *Earth* equal those of *Sun* plus twice those of *Moon*. As before, he would like to find out how many times each app was downloaded.

Let us continue calling x and y the downloads of *Sun* and *Moon* respectively, and denote by z the number of *Earth* purchases.

Now our mathematical model will be a 3×3 system, namely:

$$\begin{cases} 4x + 1.5y + 2.5z = 2470 \\ x + y + z = 972 \\ -x - 2y + z = 0. \end{cases} \tag{2.7}$$

Although the most natural form for the third piece of information of *Three apps* would be $z = x + 2y$, we have written the equation with all unknowns on the left, because in this form the system is better suited for the Cramer method and GEM.

Just as we have done for problem *Two apps*, we build the coefficient matrix A

$$A = \begin{bmatrix} 4 & 1.5 & 2.5 \\ 1 & 1 & 1 \\ -1 & -2 & 1 \end{bmatrix}$$

and the right-hand-side vector

$$\mathbf{b} = \begin{bmatrix} 2470 \\ 972 \\ 0 \end{bmatrix},$$

and call $\mathbf{x} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$ the solution vector.

Also the system of problem *Three apps* may be written in matrix form $A\mathbf{x} = \mathbf{b}$, provided we clarify how to generalise the product between A and $\mathbf{x}$ in presence of 3 components instead of 2.

Consider the general case of an $n \times n$ square matrix A, like matrix (2.4) on page 29, and a column vector

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

with n components, and let us treat $n = 3$ as special case.

The Product of an $n \times n$ Matrix with a Column Vector of Dimension n

Let A be an $n \times n$ matrix and $\mathbf{x}$ a column vector of dimension n. The product $A\mathbf{x}$ is a column vector of dimension n whose elements are computed as follows:

1 a. multiply each element of the first row of A by the element in $\mathbf{x}$ in the same position, i.e. $a_{11} \cdot x_1, a_{12} \cdot x_2, \ldots, a_{1n} \cdot x_n$;

1 b. add these values; the result

$$a_{11} \cdot x_1 + a_{12} \cdot x_2 + \ldots + a_{1n} \cdot x_n$$

is the first element of the product;

2 a. multiply each element of the second row of A by the element in $\mathbf{x}$ in the same position: $a_{21} \cdot x_1, a_{22} \cdot x_2, \ldots, a_{2n} \cdot x_n$;

2 b. add these

$$a_{21} \cdot x_1 + a_{22} \cdot x_2 + \ldots + a_{2n} \cdot x_n$$

and write it as second element of the product;

... repeat the process for rows $3, 4, \ldots, n-1$;

n a. at last, multiply every element of the nth row of A by the element of $\mathbf{x}$ in the same position: $a_{n1} \cdot x_1, a_{n2} \cdot x_2, \ldots, a_{nn} \cdot x_n$;

n b. add them together, and write

$$a_{n1} \cdot x_1 + a_{n2} \cdot x_2 + \ldots + a_{nn} \cdot x_n$$

as the nth (last) element of the product.

Matrix–vector product

The product $A\mathbf{x}$ of an $n \times n$ matrix and a column vector of dimension n is

$$A\mathbf{x} = \begin{bmatrix} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n \\ \ldots \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n \end{bmatrix}. \tag{2.8}$$

Exercises

Solve Exercise 2.3, p. 204 and Exercise 2.4, p. 205.

2.2.6 The 3 × 3 Linear System in Matrix Form

Now, as we did for problem *Two apps*, we can write the linear system of *Three apps* in the following matrix form.

The mathematical model for problem "Three apps"

Given the matrix $A = \begin{bmatrix} 4 & 1.5 & 2.5 \\ 1 & 1 & 1 \\ -1 & -2 & 1 \end{bmatrix}$ and the vector $\mathbf{b} = \begin{bmatrix} 2470 \\ 972 \\ 0 \end{bmatrix}$,

determine the vector $\mathbf{x} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$ solving the equation

$$A\mathbf{x} = \mathbf{b}.$$

The system in matrix form $A\mathbf{x} = \mathbf{b}$ represents the *mathematical model* for solving problem *Three apps*.

For this problem, too, we can point out its:

- *data*: the matrix A and the vector $\mathbf{b}$;
- *mathematical model*: the system in matrix form $A\mathbf{x} = \mathbf{b}$;
- *solution*: the vector $\mathbf{x}$.

Several problems, one model

The matrix notation allows us to represent both 2×2 and 3×3 systems by the same matrix equation $A\mathbf{x} = \mathbf{b}$.

Let us now explain how to solve the 3×3 system of problem *Three apps* using the Cramer method and GEM. We drop the substitution method because it does not let itself generalise well to systems of order bigger than two.

2.2.7 The Cramer Method for a 3 × 3 System

In order to apply the Cramer method we must be able to compute the determinant of a 3×3 matrix.

The determinant of a 3×3 matrix

Let us consider a matrix of 3 rows and 3 columns

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}.$$

Recalling that a 2×2 determinant is given by the formula

$$\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} = a_{11} \cdot a_{22} - a_{21} \cdot a_{12}$$

we define the determinant of a 3×3 matrix by the rule

$$\begin{aligned} \det(A) &= a_{11} \begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix} - a_{12} \begin{vmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{vmatrix} + a_{13} \begin{vmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{vmatrix} \\ &= a_{11}(a_{22} \cdot a_{33} - a_{32} \cdot a_{23}) - a_{12}(a_{21} \cdot a_{33} - a_{31} \cdot a_{23}) \\ &\quad + a_{13}(a_{21} \cdot a_{32} - a_{31} \cdot a_{22}). \end{aligned} \tag{2.9}$$

The Cramer method requires that we perform the following operations.

1. Compute $\det(A)$ by formula (2.9):

$$\begin{aligned} \det(A) &= 4 \begin{vmatrix} 1 & 1 \\ -2 & 1 \end{vmatrix} - 1.5 \begin{vmatrix} 1 & 1 \\ -1 & 1 \end{vmatrix} + 2.5 \begin{vmatrix} 1 & 1 \\ -1 & -2 \end{vmatrix} = \\ &= 4 \cdot 3 - 1.5 \cdot 2 + 2.5 \cdot (-1) = 6.5. \end{aligned}$$

2. Define the matrix $\hat{A}_1$, obtained by replacing the first column of A by the vector $\mathbf{b}$:

$$\hat{A}_1 = \begin{bmatrix} 2470 & 1.5 & 2.5 \\ 972 & 1 & 1 \\ 0 & -2 & 1 \end{bmatrix},$$

and compute the determinant by (2.9), to get

$$\det(\hat{A}_1) = 1092.$$

3. Define the matrix $\hat{A}_2$, obtained by replacing the second column of A by $\mathbf{b}$:

$$\hat{A}_2 = \begin{bmatrix} 4 & 2470 & 2.5 \\ 1 & 972 & 1 \\ -1 & 0 & 1 \end{bmatrix},$$

and compute its determinant by (2.9), to get

$$\det(\hat{A}_2) = 1378.$$

4. Define $\hat{A}_3$, obtained by replacing the third column of A by $\mathbf{b}$:

$$\hat{A}_3 = \begin{bmatrix} 4 & 1.5 & 2470 \\ 1 & 1 & 972 \\ -1 & -2 & 0 \end{bmatrix}$$

and compute its determinant by (2.9), to get

$$\det(\hat{A}_3) = 3848.$$

5. Compute the solution of the system

$$x = \frac{\det(\hat{A}_1)}{\det(A)} = \frac{1092}{6.5} = 168,$$

$$y = \frac{\det(\hat{A}_2)}{\det(A)} = \frac{1378}{6.5} = 212,$$

$$z = \frac{\det(\hat{A}_3)}{\det(A)} = \frac{3848}{6.5} = 592.$$

Once again David gets what he was looking for: that there were 168 downloads of *Sun*, 212 of *Moon* and 592 of *Earth*.

Even though the data and the number of variables are different from those of problem *Two apps*, the steps we have taken to solve the 3×3 system by the Cramer method are exactly the same as for the 2×2 system.

In fact, if we call n (equal to 2 or 3) the order of the system, and denote by x_k the kth component of the solution vector $\mathbf{x}$, the Cramer method can be implemented as follows:[8]

Algorithm 1 *Cramer method*

Data: the matrix A and the right-hand-side vector $\mathbf{b}$
Result: the solution vector $\mathbf{x}$
compute $\det(A)$;
for $k = 1, \ldots, n$ **do**
  build the matrix $\hat{A}_k$ by setting $\hat{A}_k = A$ and substituting the kth column of $\hat{A}_k$ with the right-hand-side vector $\mathbf{b}$;
  compute $\det(\hat{A}_k)$;
end
for $k = 1, \ldots, n$ **do**
  compute the kth component of the solution vector $\mathbf{x}$ by the formula $x_k = \det(\hat{A}_k)/\det(A)$;
end

[8] See the next subsection for the interpretation of the algorithm.

The Cramer method is an algorithm

The Cramer method is therefore an *algorithm*, in other words a systematic computational process that, by operating on the data A and $\mathbf{b}$, provides the solution $\mathbf{x}$. The algorithm ends if $\det(A) \neq 0$ (division by zero is not defined in $\mathbb{R}$); this is, to all effects, a necessary and sufficient condition for the solution of the linear system $A\mathbf{x} = \mathbf{b}$ to exist and be unique.

Algorithm 1 remains valid also for systems of order $n > 3$ as long as we have a formula for the determinant.[9]

2.2.8 *for* Loops in Algorithms

The expression

for $k = 1, \ldots, n$ **do**
  build the matrix $\hat{A}_k$ by setting $\hat{A}_k = A$ and substituting the kth column of $\hat{A}_k$ with the vector $\mathbf{b}$;
  compute $\det(\hat{A}_k)$;
end

is an example of a *for loop*. It means the following: "build the matrix $\hat{A}_k$ (by setting $\hat{A}_k = A$ and substituting the kth column of $\hat{A}_k$ by the vector $\mathbf{b}$) and compute $\det(\hat{A}_k)$ for every integer value k going from 1 to n".

Therefore the loop

for $k = 1, \ldots, 3$ **do**
  build the matrix $\hat{A}_k$ by setting $\hat{A}_k = A$ and substituting the kth column of $\hat{A}_k$ with the vector $\mathbf{b}$;
  compute $\det(\hat{A}_k)$;
end

is equivalent to the following instructions (to be executed precisely in this order):

($k = 1$)
build the matrix $\hat{A}_1$ by setting $\hat{A}_1 = A$ and substituting the *first* column of $\hat{A}_1$ with the vector $\mathbf{b}$;
compute $\det(\hat{A}_1)$;

($k = 2$)
build the matrix $\hat{A}_2$ by setting $\hat{A}_2 = A$ and substituting the *second* column of $\hat{A}_2$ with the vector $\mathbf{b}$;
compute $\det(\hat{A}_2)$;

($k = 3$)
build the matrix $\hat{A}_3$ by setting $\hat{A}_3 = A$ and substituting the *third* column of $\hat{A}_3$ with the vector $\mathbf{b}$;
compute $\det(\hat{A}_3)$.

[9] There are several such formulas for the determinant of $n \times n$ matrices when $n > 3$, but all of them require defining further mathematical tools that go beyond the scope of this book.

The instructions written between the start of the loop (the word **do**) and its end (the word **end**) make up what is known as the *body of the loop*.

Here is another example:

for $k = 3, \dots, 6$ **do**
  compute $k^2 - 1 + k$;
end

This loop is the same as this sequence of single instructions:

compute $3^2 - 1 + 3$ (corresponding to $k = 3$);
compute $4^2 - 1 + 4$ (corresponding to $k = 4$);
compute $5^2 - 1 + 5$ (corresponding to $k = 5$);
compute $6^2 - 1 + 6$ (corresponding to $k = 6$).

The for loop

for $k = 5, \dots, 2$ *with negative increment* **do**
  compute $k^2 - 1 + k$;
end

is equivalent to

compute $5^2 - 1 + 5$ (corresponding to $k = 5$);
compute $4^2 - 1 + 4$ (corresponding to $k = 4$);
compute $3^2 - 1 + 3$ (corresponding to $k = 3$);
compute $2^2 - 1 + 2$ (corresponding to $k = 2$).

In this last example the value of k decreases by one at each iteration, as opposed to increasing as we had previously.

2.2.9 Variable Assignment in an Algorithm

If we inspect Algorithm 1 carefully, we may note that the determinant of matrix A is computed by the first instruction, but then it gets used only during the final loop to compute the components x_k of the solution vector. In the meantime we are performing many other computations.

If we did all calculations prescribed by the algorithm by hand, we would quite likely be forced to write down the value of the determinant of A on the side, so not to forget it whilst doing other computations.

Since computers are even more forgetful than us (there is no way they will retain a result unless we force them to store it immediately), if we want a computer to execute our algorithm, suitably translated into a program, we must tell it to save the value $\det(A)$ in a variable.[10]

Otherwise put, we must execute the operation called *variable assignment* by replacing the instruction

compute $\det(A)$;

[10] A *variable* in an algorithm or a program is, effectively, a box ready to contain anything we place in it.

with the instruction
$d = \det(A)$.
In this way the algorithm computes the determinant of A and stores it in the variable called d, so that it can be recovered and used at a later stage.

Beware:

we	WRITE	$d = \det(A)$,
and	NOT	$\det(A) = d$,

because the variable in which we store the result must always be placed to the left of the equal sign (this is a convention adopted in programming languages and algorithms).

Let us now consider the following algorithm:

$s = 0$;
for $k = 1, \ldots, 10$ **do**
| $s = s + k$;
end

and let us try to understand what the instruction inside the body of the loop means.

Some might be tempted to cross out the s on either side of the equal sign, but this would not make any sense! Remember that these are not equations, but the instructions of the algorithm!

The instruction $s = s + k$ means the following:

"take the value of s, add it to the value of k and save the final result again in the variable s".

More generally, the idea is: *compute what is required on the right, and save the result in the variable written on the left*.

The above short algorithm performs these steps:

- it assigns to s the value zero
- it initiates the loop in the variable k by setting $k = 1$
- it takes the value saved in s (that is, 0), it adds it to k (that is, 1) and stores the result in s, erasing the previous value, so that now we have $s = 1$
- the loop moves on to $k = 2$
- it takes the value saved in s (i.e. 1), it adds it to $k = 2$ and stores the result in s (erasing the previous value), so now $s = 3$
- the loop continues with $k = 3$
- it takes the value saved in s (that is, 3), it adds it to $k = 3$ and stores the result in s (erasing the previous value), so $s = 6$
- and so on and so forth.

At the end of the iteration the variable s will contain the value 55.

2.2.10 The Gauss Elimination Method (GEM) for a 3 × 3 System

As we highlighted for 2 × 2 systems, GEM consists of two phases: the first is the reduction and the second the back-substitution. To carry out the reduction we must

remember that the final goal is to achieve a system suitable for the back-substitution: starting from the value of the last component of the solution vector $\mathbf{x}$, which we can compute by solving the last equation, we have to use the equations of the system one by one, backwards, in order to determine all the unknown components.

In general, to carry out the reduction we must make sure that:

- the last equation contains only the last unknown, so that we may compute the latter directly;
- the second-last equation contains only the last two unknowns;
- the third-last contains only the last three unknowns;
- and so on...
- the second equation contains all variables but the first;
- the first equation contains all variables.

Let us show how to implement this on the 3×3 system (2.7) of page 34.

1. *Reduction of the system.*
   a. We wish to eliminate the unknown x from the second and third equations of system (2.7).

      From the second equation of system (2.7) we subtract the first one multiplied by $\frac{1}{4}$ (this is the multiplier associated with the second equation, needed to eliminate x) so to obtain a new equation without x:

$$\begin{array}{rl} \text{second equation:} & x + y + z = 972 \\ -\frac{1}{4}\cdot\text{(first equation):} & -\frac{1}{4}\left(4x + \frac{3}{2}y + \frac{5}{2}z = 2470\right) \\ \hline & \frac{5}{8}y + \frac{3}{8}z = \frac{709}{2}. \end{array}$$

      From the third equation of system (2.7) we subtract the first one multiplied by $-\frac{1}{4}$ (the multiplier associated with the third equation to get rid of x), again for the purpose of getting a new equation without x:

$$\begin{array}{rl} \text{third equation:} & -x - 2y + z = 0 \\ -\left(-\frac{1}{4}\right)\cdot\text{(first equation):} & -\left(-\frac{1}{4}\right)\left(4x + \frac{3}{2}y + \frac{5}{2}z = 2470\right) \\ \hline & -\frac{13}{8}y + \frac{13}{8}z = \frac{1235}{2}. \end{array}$$

Then we substitute the second and third rows of system (2.7) and write these new equations in their place, in the same order they were obtained:

$$\begin{cases} 4x + \frac{3}{2}y + \frac{5}{2}z = 2470 \\ 0x + \frac{5}{8}y + \frac{3}{8}z = \frac{709}{2} \\ 0x - \frac{13}{8}y + \frac{13}{8}z = \frac{1235}{2} \end{cases} \tag{2.10}$$

b. Now we want to eliminate y from the third equation just obtained. For that, we subtract from the third equation of system (2.10) the second equation times $-\frac{13}{5}$ (the multiplier associated with the third equation) to get rid of y:

$$\begin{array}{rrl} \text{third equation:} & -\frac{13}{8}y + \frac{13}{8}z & = \frac{1235}{2} \\ -\left(-\frac{13}{5}\right) \cdot \text{(second equation):} -\left(-\frac{13}{5}\right) & \left(\frac{5}{8}y + \frac{3}{8}z\right. & \left.= \frac{709}{2}\right) \\ \hline & \frac{13}{5}z & = \frac{7696}{5}. \end{array}$$

Finally, we substitute the third row of system (2.10) with this last equation, to get:

$$\begin{cases} 4x + \frac{3}{2}y + \frac{5}{2}z = 2470 \\ 0x + \frac{5}{8}y + \frac{3}{8}z = \frac{709}{2} \\ 0x + 0y + \frac{13}{5}z = \frac{7696}{5} \end{cases} \tag{2.11}$$

which is still equivalent to system (2.7) of page 34.

2. *Back-substitution of variables.*
Now the true substitution begins, together with the proper computation of the variables (vector components). Let us start from the last equation of system (2.11) and work backwards until we reach the first one:
   a. from the last equation we have $z = \frac{7696}{13} = 592$;
   b. substitute this z in the second equation of system (2.11)

$$\frac{5}{8}y + \frac{3}{8} \cdot 592 = \frac{709}{2}$$

   to find $y = 212$;
   c. substitute the values of y and z in the first equation of system (2.11)

$$4x + \frac{3}{2} \cdot 212 + \frac{5}{2} \cdot 592 = 2470$$

   to obtain $x = 168$.

GEM is an algorithm

As for the Cramer method, also GEM may be formulated for systems of *any order* n. Said otherwise: GEM, too, is an algorithm.

Implementing it, however, is less immediate than implementing the Cramer algorithm.

In order to write down the GEM algorithm we shall call $x_1, x_2, \ldots, x_n$ the components of the solution vector $\mathbf{x}$. The following coding of GEM, in which the commands are written in the spoken language, generalises to systems of order $n \geq 2$ the earlier instructions for $n = 2$ (page 32) and $n = 3$ (page 41). We suggest the reader go over those computations following the steps indicated below, and refer to page 39 for the description of the *for loops*.

Algorithm 2 *Gauss Elimination Method (GEM)*

Data: the linear system of n equations in n variables
Result: solutions $x_1, x_2, \ldots, x_n$

```
% reduction
```

for $k = 1, \ldots, n-1$ **do**
- eliminate the unknown x_k from all equations after the kth one, i.e.:
- **for** $i = k+1, \ldots, n$ **do**
  - – compute the multiplier m_{ik} = ratio of the coefficient of x_k on row i over the coefficient of the same variable x_k on row k;
  - – substitute row i with the difference between row i itself and m_{ik} times row k
- **end**

end

```
% back-substitution of the variables
```

compute x_n from the last equation;
for $i = n-1, n-2, \ldots, 2, 1$ *(the counter i decreases by 1 each time)* **do**
- substitute in row i all variables already computed ($x_n, x_{n-1}, \ldots, x_{i+1}$) and compute x_i;

end

Exercises

Solve Exercise 2.5, p. 205.

Pivoting

We must point out that in some situations, even if the linear system is determined (there exists a unique solution $\mathbf{x}$), Algorithm 2 might not stop and reach an end. This happens whilst computing the multipliers m_{ik} if we happen to force a division by zero. In this case GEM can be modified by the introduction of so-called *pivoting*, a suitable shuffle in the order the equations appear in the system. The study of this

circumstance, which never occurs for several types of matrices, would lead us way beyond the present purpose.

2.2.11 We Reached 100!

Now we are all geared up for the following problem.

Problem "100 apps"

David will certainly not stop there. He creates a start-up with three mates and the following year they write 97 more apps together, for a grand total of 100! Apart from knowing the overall number of downloads and the total revenue, they manage to get hold of further 98 pieces of information that guarantee they can find out the downloads of each app. So David and his friends set out to calculate these numbers exactly.

The mathematical model for problem "100 apps"

The *mathematical model* is now a system of 100 linear equations in 100 unknowns, so a system of order $n = 100$ which can still be written in matrix form as

$$A\mathbf{x} = \mathbf{b}.$$

As before, we have:

- the *data of the problem*: the matrix A (100 rows and 100 columns, containing the coefficients of the 100 variables present in the 100 equations) and the vector $\mathbf{b}$ (a column vector of 100 components corresponding to the right-hand-side terms of the equations);
- the *mathematical model*: the system in matrix form $A\mathbf{x} = \mathbf{b}$;
- the *solution*: the vector $\mathbf{x}$ (a column vector of 100 components, each representing the number of downloads of one app).

To solve system $A\mathbf{x} = \mathbf{b}$ we may use the Cramer method or GEM, given that we are able to generalise both to a system of arbitrary order n.

But doing the computations by hand for a linear system of order 100, or order n in general, might involve a staggering number of elementary operations (sums, subtractions, multiplications and divisions).

2.2.12 How Many Operations Must We Do?

To quantify the number of operations required by the Cramer method we must first define the *factorial* of a natural number n, denoted by $n!$:

$$0! = 1,$$
$$n! = 1 \cdot 2 \cdot 3 \cdots (n-1) \cdot n, \qquad \text{for } n > 0.$$

The product $1 \cdot 2 \cdot 3 \cdots (n-1) \cdot n$ can be written in a more concise way as follows:

$$1 \cdot 2 \cdot 3 \cdots (n-1) \cdot n = \prod_{k=1}^{n} k$$

where $\prod_{k=1}^{n} k$ is read: "product of all integers k from 1 to n."

It can be proved that to solve a linear system of order n one needs:

- about $3 \cdot (n+1)! = 3 \cdot (n+1) \cdot n \cdot (n-1) \cdots 3 \cdot 2$ operations if one uses the Cramer method;
- $\frac{2}{3}n^3 + \frac{3}{2}n^2 - \frac{7}{6}n$ operations with GEM.

This estimate is a bit complicated to explain here and we ask to take it for granted.

To truly understand how many operations are at stake, in Table 2.3 we wrote down the values of the above expressions for some choices of $n \geq 2$.

It is obvious that solving by hand a system of order 10 would require more time than we have, and in all likelihood we would make some mistakes before reaching the end... So imagine a 100×100 system!

Someone might say that solving a system of order $n = 100$ is only the whim of mathematicians, but it is really not so. In practical applications, for instance to make a weather forecast, one needs to solve linear systems with several *millions of equations and variables*, that is, a system with $n = 10^6,\ 10^7,\ 10^8$!

Table 2.3 The number of elementary operations (sums, subtractions, multiplications or divisions) required by the Cramer method and by GEM for solving a linear system of order n

Order of system n	Elementary operations with Cramer $3(n+1)!$	Elementary operations with GEM $\frac{2}{3}n^3 + \frac{3}{2}n^2 - \frac{7}{6}n$
2	18	5
3	72	19
4	360	46
5	2160	90
10	119750400	705
20	$\simeq 10^{20}$	5510
...	...	...
100	$\simeq 3 \cdot 10^{160}$	671550

2.2.13 The Computer Comes into Play

Without the shadow of a doubt the most sensible way to solve problem *100 apps* and get a result in a reasonable time is to use a computer.

Since both the Cramer method and GEM are algorithms, with the help of a programming language we may reduce them to *programs* (structured sequences of instructions written in some programming language) that the computer will execute.

To this end, though, we should caution against executing a program on the computer that implements the Cramer method on a 100×100 system.

For instance, the fastest computer on earth at the date of November 2019, the Summit IBM at Oak Ridge National Laboratory in the U.S. (see Fig. 2.10), performs about $1.5 \cdot 10^{17}$ elementary operations per second:[11] if it were to use the Cramer method it would take it around 10^{127} billion years to compute the solution of our system of order $n = 100$.

This fact highlights another aspect of scientific computing: not every numerical method we use to deal by hand with problems of small order ($n = 2$ or $n = 3$) are efficient when n becomes significantly larger. The Cramer method is just one example.

Exercises

Solve Exercise 2.6, p. 207.

Fig. 2.10 Summit IBM, the fastest computer in the world in November 2019. (©Carlos Jones, ORNL, Oak Ridge (US), 2019)

[11] To get a concrete idea: it is as if all 7 billion human beings on the planet could, each, do 20 million operations in one second, simultaneously!

We have found an example of a *beautiful and very general* mathematical formula that is completely *useless* to handle linear systems of any order that is not tiny.

Luckily GEM is a much more economical numerical method than the Cramer method, as Table 2.3 confirms. To solve the same linear system of order n, GEM requires much fewer operations than the Cramer method:

$$\frac{2}{3}n^3 + \frac{3}{2}n^2 - \frac{7}{6}n \text{ for GEM against about } 3(n+1)! \text{ for Cramer.}$$

In particular, when $n = 100$, we have

$$\frac{2}{3}n^3 + \frac{3}{2}n^2 - \frac{7}{6}n \simeq 7 \cdot 10^5,$$

which means that solving a system of order 100 on our laptop (which is capable of about 10^7 elementary operations per second) requires *less than a second*.

Computational cost

The *computational cost* of an algorithm is measured in terms of the elementary operations the computer must do to execute the instructions in the algorithm. The smaller the computational cost of an algorithm, the less time we spend in front of the screen waiting for an answer.

2.2.14 GEM, Too, Adapts to the Computer

Algorithm 2 on page 44 is not written in the most suitable form for translation into a computer program. It is better to rewrite it using matrices and vectors and swap the instructions, initially expressed in common language, with algebraic operations.

We are aware that the following algorithm might seem a little incomprehensible at first. The first time we saw it we were puzzled, too. Not every one of us is Gauss, but perhaps there is one reader out there, who knows, . . .

***Algorithm 3** GEM with matrices and vectors*

```
Data: the matrix A and the right-hand-side vector b
Result: the solution vector x
% reduction
for k = 1, ..., n − 1 do
    for i = k + 1, ..., n do
        m_ik = a_ik / a_kk;
        for j = 1, ..., n (this loop may be shortened by taking only
          j = k + 1, ..., n) do
            a_ij = a_ij − m_ik a_kj;
        end
        b_i = b_i − m_ik b_k;
    end
end
% back-substitution of variables
x_n = b_n / a_nn;
for i = n − 1, n − 2, ..., 2, 1 do
    s = 0;
    for j = i + 1, ..., n do
        s = s + a_ij x_j;
    end
    x_i = (b_i − s) / a_ii;
end
```

First of all observe that in the reduction phase the loop *"for* $j = 1, \ldots, n$*"* can be replaced by the loop *"for* $j = k+1, \ldots, n$*"* without changing the final result. In fact we know that when the counter j lies between 1 and k, the matrix element a_{ij} obtained by executing the instruction

$$a_{ij} = a_{ij} - m_{ik}a_{kj}$$

is zero (because of how GEM is set up) for every i between $k+1$ and n, and it does not get used any longer in the algorithm. Therefore not simplifying would only waste operations and time.

Now let us examine the other instructions, recalling we have already met *for loops* on page 39 and the operations of variable assignment on page 40.

In particular, the instruction $m_{ik} = a_{ik}/a_{kk}$ means:

"divide a_{ik} (the element in A on row i and column k) by a_{kk} (element in A on row k and column k), and save this value in the variable called m_{ik}".

Similarly, the instruction $x_n = b_n/a_{nn}$ means:

"divide b_n (nth component of the vector $\mathbf{b}$) by a_{nn} (element in A on row n and column n) and save the value in the nth component of vector $\mathbf{x}$".

On the other hand the instruction $s = 0$ just means we store the value 0 in the variable called s.

The instructions $a_{ij} = a_{ij} - m_{ik}a_{kj}$, $b_i = b_i - m_{ik}b_k$ and $s = s + a_{ij}x_j$ in Algorithm 3 are variable assignments as well (more precisely, updating instructions), as we saw on page 40.

In particular, the instruction $s = s + a_{ij}x_j$ means:

"take the value of s, add it to the product of a_{ij} by x_j and save the final result again in the variable s".

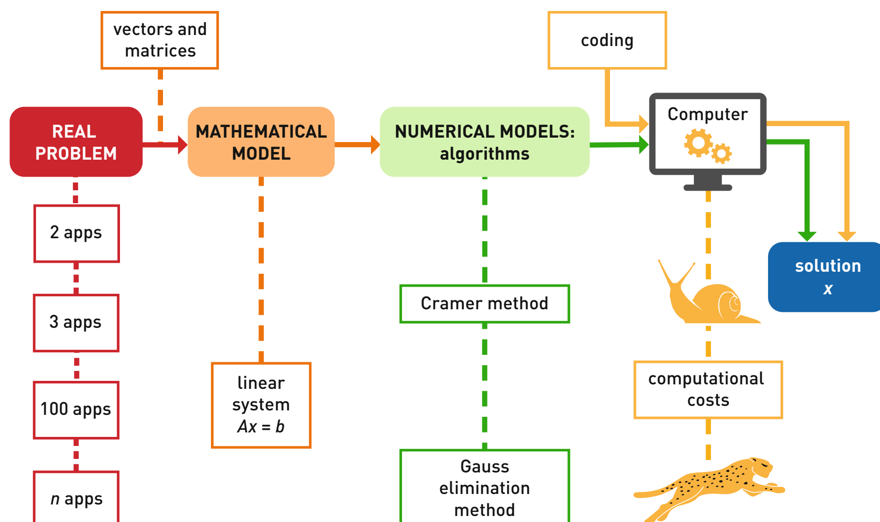


Fig. 2.11 The conceptual map of Sect. 2.2

Analogously, $a_{ij} = a_{ij} - m_{ik}a_{kj}$ means:

"take the value of a_{ij}, subtract from it the product of m_{ik} and a_{kj}, and save the result of this operation again in the variable a_{ij}".

Eventually, $b_i = b_i - m_{ik}b_k$ means:

"take the value of b_i, subtract the product of m_{ik} and b_k, and save this result in the variable b_i".

All three instructions obey the rule: *compute what is demanded on the right of the equal sign and save the result in the variable on the left.*

Exercises

Solve Exercise 2.7, p. 207.

In Chap. 8, after reviewing the basic Octave instructions, we shall translate Algorithm 3 in a proper program, which we will then use to solve linear systems.

The flowchart in Fig. 2.11 sums up the path that has led us to the solution of the problems of this section.

2.3 What We Have Learnt

Mathematical Tools

1. A *trapezoid* given by the graph of a function is the region in the plane delimited by the x-axis, two vertical lines and the graph of the function itself.
2. *Vectors* and *matrices* are arrays of numbers organised in rows and columns.
3. An *algorithm* is a systematic computational procedure, inside which there may be *for loops* and operations called *variable assignments*.

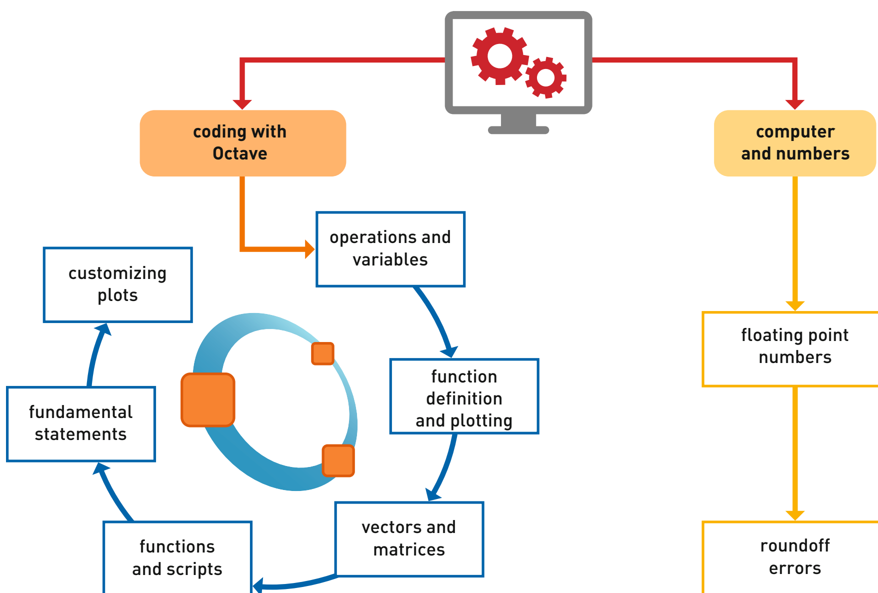
Models

1. The mathematical model serving the purpose of estimating the number of girls with height within a certain range is translated into the calculation of the *area of a trapezoid*.
2. Systems with an arbitrary number n of linear equations in as many unknowns can be represented by one *matrix equation* involving vectors and matrices.

Numerical Methods

1. To approximate the area of a trapezoid we have used the *midpoint formula*. Quadrature formulas provide approximations of integrals.
2. To solve a linear system we have introduced the *Cramer method* and the *Gauss Elimination Method (GEM)*. Both are *algorithms* and can be translated into *programs* for the computer.
3. The Cramer method is a simple example of an algorithm: it is useless when the order of the system is bigger than 3, for it requires to execute a very large number of operations.
4. On the contrary, GEM is slightly more complicated to formulate but more efficient than the Cramer method, since it needs a considerably smaller number of operations (about n^3 operations, against $3(n+1)!$ operations for Cramer).
5. The *computational cost* of an algorithm or a numerical method is quantified in terms of the elementary operations (sums, subtractions, multiplications and divisions) the computer must do to complete the algorithm.

3 Reckoning with the Computer



INGREDIENTS	Section: functions, matrices... and the desire to surpass mathematics *done by hand*. Section 3.2: the set of real numbers $\mathbb{R}$, elementary properties of algebraic operations.
WHAT WE LEARN	Section 3.1: basics on Octave programming. Section 3.2: the finite arithmetic of computers, origin of roundoff errors.
PREREQUISITES	Section 3.1: none. Section 3.2: Sect. 3.1.

When a numerical model generates an algorithm, the latter must be implemented in a program by means of a programming language. Among the many languages

© Springer Nature Switzerland AG 2020

A. Quarteroni, P. Gervasio, *A Primer on Mathematical Modelling*, UNITEXT 121,
https://doi.org/10.1007/978-3-030-44541-6_3

available we shall focus on Octave, because on one hand it is a simple language, and on the other it is particularly efficient for scientific computing.

We shall indicate on page margins the commands introduced throughout.

3.1 Programming with Octave

Octave, also known as GNU Octave,[1] is a freely distributed software that can be downloaded from www.octave.org. Once Octave is installed on our computer, every time we launch the program a window opens on the desktop like the one shown in Fig. 3.1.

The window is divided in sub-windows that allow to execute commands, view the output, manage files, visualise the content of the variables and of the working memory.

The main window is the *command window*, which is characterised by the presence of the *prompt*:

```
>>
```

This is the place in which we type commands and instructions.

Let us see how to work with Octave by looking at simple problems.

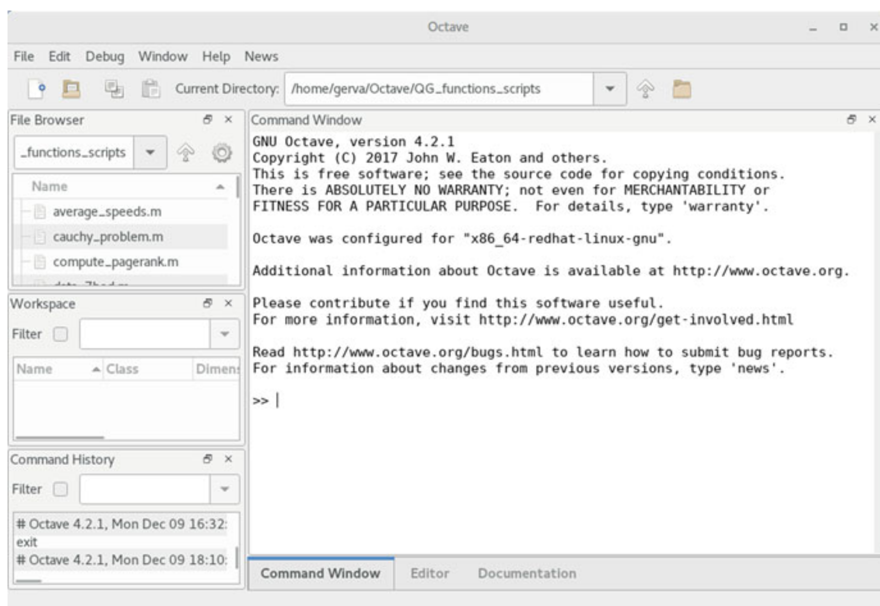


Fig. 3.1 The Octave working environment

[1] John W. Eaton, David Bateman, Søren Hauberg, Rik Wehbring (2018). GNU Octave version 4.4.1 manual: a high-level interactive language for numerical computations. URL https://www.gnu.org/software/octave/doc/v4.4.1/

3.1.1 Arithmetic Operations and Variables

Exercise: Computation of a numerical expression

Compute the real number

$$a = \frac{3 + 5^3 - \frac{2}{3}}{4.25(5 + 2^4)}$$

Solution. Type the instruction

```
a=(3+5^3-2/3)/(4.25*(5+2^4))
```

in the command window, right after the prompt », and press Enter. This will give the result

```
a =   1.4267
```

Octave has computed the result and has stored it in the variable[2] named `a`. □

When writing the numerical expression we have followed certain *rules*:

1. the comma of decimal numbers is expressed by a full stop;
2. arithmetic operations are denoted by the following symbols (or, operators):

`^ * / + -`

^	raising to a power
*	product
/	quotient
+	sum
-	difference

3. the standard rules are obeyed when performing a sequence of operations: first powers, then multiplications and divisions (from left to right), and at last additions and subtractions (from left to right);
4. in order to modify the above implicit hierarchy one can use round brackets, but only those.

Octave displays the result with 4 decimal digits only (after the decimal point), even if it actually works with 15 digits. To see all 15 digits of a number we must type the command

`format`

```
format long
```

[2] As we said in Chap. 2, a *variable* inside an algorithm or program is in practice a box that can contain whatever we decide to store in it.

and press Enter. This does not produce an output, but from this moment onward we force Octave to display all results in the 15-digit *format*. So if we type

```
a
```

and press Enter, we obtain

```
a = 1.42670401493931
```

because Octave returns the result computed while solving problem *Computation of a numerical expression*, which was stored in variable a.

To go back to the 4-decimal-digit format it is enough to type

```
format short
```

From now on we shall implicitly assume that after each command the Enter button is pressed.

Names of Octave variables

The *names of variables* in Octave are sequences of at most 63 alphanumerical characters, the first of which cannot be a number.

Certain special characters are allowed, such as the *underscore* _, while others are forbidden: no spaces, no operation symbols, no *keywords* of the language such as for, end, if, while...

Furthermore, Octave is *case sensitive*, so it treats the upper case and the lower case differently. For example a and A are distinct variables and are allowed to contain different numbers.

Suppose we are not interested, when performing an operation or assigning a value to a variable, in viewing the result or the value immediately. Then we can end the instruction with a semicolon ; . For instance, the instruction

;

```
b=3.42/72;
```

will not display anything, yet Octave has performed the operation and it has saved the result in the variable b.

If we want to see the content of b (the number that was saved in b) it will be enough to type

```
b  % without the semicolon
```

and Octave will answer:

```
b = 0.047500
```

Anything that comes after the command % (before the end of the line) is considered a comment by Octave, and hence not a command.

%

Observe that when we perform an operation without assigning it to a variable, Octave stores the result of the operation in a default variable called ans. For example the instruction

ans

```
3.42/72
```

produces

```
ans = 0.047500
```

The content of the variable ans changes as soon as we do another operation without saving it in a specific variable. So if we now write

```
2/3.56
```

Octave produces

```
ans = 0.56180
```

because Octave has wiped the ans variable from the value 0.047500 and it has stored 0.56180 in it (remember a variable is much like a box: the last operation has emptied ans of its previous content and has put in it something else).

3.1.2 Mathematical Functions and Their Graphical Representation

Exercise: Plotting mathematical functions

Plot the functions
$f(x) = 1 - \frac{x}{2}$ and $g(x) = \sqrt{2x+5} - 2$ over the interval $I = [-2, 6]$.

Solution. We type the following Octave instructions in the command window (you can avoid to write the comments):

```
f=@(x)1-x/2; % defines the function f
g=@(x)sqrt(2*x+5)-2; % defines the function g
figure(1);  % chooses the graphical window number 1 to plot
clf      % wipes the graphical window (if it was used before)
fplot(f,[-2,6],'r-') % plots the function f over the interval
       % [-2,6] in red with a solid line
hold on    % keeps this graph and plots the next one
fplot(g,[-2,6],'b--') % plots the function g over the interval
       % [-2, 6] in blue with a dashed line
xlabel('x')  % assigns the label 'x' to the x-axis
ylabel('y')  % assigns the label 'y' to the y-axis
legend('f(x)=1-x/2','g(x)=sqrt(2x+5)-2') % displays the
       % expression of the plotted functions
axis equal  % displays the graph using the same unit in x and y
grid on    % plots a grid over the graph
```

The outcome picture is represented in Fig. 3.2. □

Now we shall explain the syntax of the commands involved, so that you may use them in other situations.

The syntax for defining a mathematical function f is

f=@(x)...

```
f=@(x)expression
```

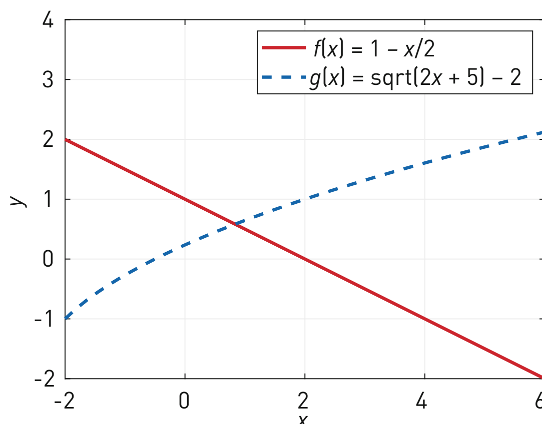


Fig. 3.2 The output of problem *Plotting mathematical functions*

where `x` is the independent variable on which the function acts, while `expression` is the actual mathematical expression of the function, for example `1-x/2` or `sqrt(2*x+5)-2`.

`f` is a new type of Octave variable, called *function handle*.

To define the function handle `g` we have used the mathematical function `sqrt` that computes the square root. If, instead, we want to define the function $h(x) = e^x$, we should type the command

`sqrt` `exp`

```
h=@(x)exp(x)
```

Table 3.1 shows the most common mathematical functions and the corresponding Octave names.

The command `fplot` allows to plot a mathematical function. Using the instruction

`fplot`

```
fplot(f,[a,b])
```

we can plot the function `f` over the interval $[a, b]$. Inside the round brackets we may add further parameters - separated by commas - to specify the colour and type of line to be used.

Table 3.1 The main mathematical functions and their Octave names

Mathematical functions	Octave commands
$\sqrt{x}$	`sqrt(x)`
$\|x\|$	`abs(x)`
$\sin(x)$, $\cos(x)$, $\tan(x)$	`sin(x)`, `cos(x)`, `tan(x)`
e^x, $\log_e(x)$, $\log_{10}(x)$	`exp(x)`, `log(x)`, `log10(x)`
$\arcsin(x)$, $\arccos(x)$, $\arctan(x)$	`asin(x)`, `acos(x)`, `atan(x)`
roundoff of x	`round(x)`, example: `round(3.6)=4`
integer part of x	`fix(x)`, example: `fix(3.6)=3`

For example the instruction

```
fplot(f,[-2,6],'r-')
```

plots the function in a red solid line, while the instruction

```
fplot(f,[-2,6],'b--');
```

draws a blue dashed line.

For the list of all possible options of an Octave command it suffices to invoke the command `help`. By typing

help

```
help fplot
```

for instance, we can discover the features of the command `fplot`.

The `xlabel('xxx')` and `ylabel('yyy')` commands add a description of the axes. More precisely, they attribute the string[3] `'xxx'` to the horizontal axis and `'yyy'` to the vertical axis.

xlabel
ylabel

At last, the `legend('text1', 'text2', ...'textn')` command allows to specify a legend in the picture with one line for each graph. The lines in the `legend` command must be typed in the same order of execution of the corresponding functions.

legend

3.1.3 Vectors and Matrices

If n and m are two positive integers, a matrix A with n rows and m columns is a table of $n \times m$ real numbers a_{ij} where $i = 1, \ldots, n, \;\; j = 1, \ldots, m$:

$$A = \begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1m} \\ a_{21} & a_{22} & \ldots & a_{2m} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \ldots & a_{nm} \end{bmatrix}$$

(see Chap. 2, Sect. 2.2 for the definition of matrices and vectors). We shall also write more compactly $A = (a_{ij})$. When $n \neq m$ the matrix is called *rectangular*, while if $n = m$ it is called a *square matrix* of order n.

A matrix with just one column is a *column vector*, and a matrix with one row only is a *row vector* (cf. Chap. 2).

In Octave matrices and vectors are generically referred to as *arrays*.

[3] A *string* is a sequence of alphanumerical characters enclosed in single quotes.

Exercise: Definition of a matrix and two vectors in Octave

Assign to the variable A the matrix

$$A = \begin{bmatrix} 4 & 1.5 & 2.5 \\ 1 & 1 & 1 \\ -1 & -2 & 1 \end{bmatrix}$$

and to the variables b, c the vectors

$$\mathbf{b} = \begin{bmatrix} 2470 \\ 972 \\ 0 \end{bmatrix}, \qquad \mathbf{c} = \begin{bmatrix} -3 & \sqrt{2} & -2.56 \end{bmatrix}.$$

A=[ ]

Solution. To assign to the variable A the matrix A we type the instruction

```
A=[4,1.5,2.5; 1,1,1; -1,-2,1] % rows are separated by semicolons
```

Octave produces the output

```
A =
  4.0000  1.5000  2.5000
  1.0000  1.0000  1.0000
  -1.0000  -2.0000  1.0000
```

The instruction

```
b=[2470; 972; 0] % components are separated by semicolons
```

produces the column vector

```
b =
    2470
     972
       0
```

while the command

```
c=[-3,sqrt(2),-2.56]% components are separated by commas
```

returns the row vector

```
c =
  -3.0000  1.4142  -2.5600
```

□

To define arrays the following rules should be respected:

1. all elements in an array are delimited by square brackets;
2. all elements of a matrix are typed row-by-row, from left to right on each row;
3. commas separate the elements on a row;
4. semicolons separate the elements on successive rows;
5. the components of a column vector (like **b**) are separated by semicolons (they belong on different rows);

6. the components of a row vector (like **c**) are separated by commas (they appear on the same row).

Note that to separate the elements of one single row we could have used a blank space instead of a comma: the instructions

```
c=[-3,sqrt(2),-2.56]
```

and

```
c=[-3 sqrt(2) -2.56]
```

are equivalent.

Exercise: Reading the elements of an array

Let A, b and c be the arrays defined in the previous exercise. Display on the screen:

- the elements a_{23}, b_2 and c_3;
- row 2 of matrix A;
- column 3 of matrix A.

Solution. To display a_{23} let us type:

```
A(2,3)
```

and the answer in Octave is:

```
ans = 1
```

Similarly, to display b_2 we write

```
b(2)
```

and the answer is

```
ans = 972
```

while for c_3 we type

```
c(3)
```

and Octave gives

```
ans = -2.5600
```

To read and display the second row of A (the elements appearing on the second row and on each column), the command to use is:

A(2,:)

```
A(2,:)
```

and Octave gives:

```
ans =
   1   1   1
```

A(:,3)

Finally, to display the third row of A (the elements on column three, one for each row), the command to use is:

```
A(:,3)
```

Answer:

```
ans =
  2.5000
  1.0000
  1.0000
```

□

To read the elements of an array we have followed these rules:

1. to read the content of a component of an array and display it on the screen, use round brackets;
2. to read the content of a matrix row and display it use round brackets, the specified row index, and the character **:** (colon) in place of the column index;
3. to read the content of a matrix column and display it use round brackets, and the character **:** (colon) in place of the row and the column indices.

Operations with Matrices

We can define a number of elementary operations on matrices.

1. If $A = (a_{ij})$ is an $n \times m$ matrix, then $A^T = (a_{ji})$ is the transpose matrix of A, obtained by swapping the elements a_{ij} and a_{ji}. The transpose is an $m \times n$ matrix. For example

$$A = \begin{bmatrix} 2 & 4 \\ -1 & 0 \\ 2 & -3 \end{bmatrix}, \qquad \text{and } A^T = \begin{bmatrix} 2 & -1 & 2 \\ 4 & 0 & -3 \end{bmatrix}.$$

   The Octave instructions to store the matrix A in the variable `A`, build the transpose and store it in the variable `At` are:

```
A=[2, 4; -1, 0; 2, -3];
At=A'
```

'

   hence, the single-quote mark `'` performs the operation of transposition.
2. Let $A = (a_{ij})$ and $B = (b_{ij})$ be $n \times m$ matrices. The *sum* of A and B is the matrix C with elements $c_{ij} = a_{ij} + b_{ij}$ (we add the elements in the same position in the two matrices); for instance, if

$$A = \begin{bmatrix} 2 & 4 \\ -1 & 0 \\ 2 & -3 \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} 1 & -2 \\ 5 & 3 \\ -1 & 4 \end{bmatrix},$$

then

$$C = A + B = \begin{bmatrix} 3 & 2 \\ 4 & 3 \\ 1 & 1 \end{bmatrix}.$$

The Octave instructions to do this sum read:

```
A=[2, 4; -1, 0; 2, -3];
B=[1, -2; 5, 3; -1, 4];
C=A+B
```

3. The *product* of a matrix A by a real number λ is the matrix $B = \lambda A$ with elements $b_{ij} = \lambda a_{ij}$ (each element of A is multiplied by λ); for instance, with

$$A = \begin{bmatrix} 3 & 1 \\ -2 & 4 \end{bmatrix} \quad \text{and} \quad \lambda = 2,$$

we have

$$B = \lambda A = \begin{bmatrix} 6 & 2 \\ -4 & 8 \end{bmatrix}.$$

The Octave instructions for this product are:

```
A=[3, 1; -2, 4];
lambda=2;
B=lambda*A
```

4. The *product of a square matrix A (n rows and n columns) and a column vector* $\mathbf{x}$ *(of n components)* is a column vector $\mathbf{c}$ of n components defined in this way (see Chap. 2, Sect. 2.2):

$$c_i = a_{i1}x_1 + a_{i2}x_2 + \ldots + a_{in}x_n, \qquad \text{for every } i = 1, \ldots, n.$$

The elements in A appearing in the formula all have row index i (the same as c_i). For example, if

$$A = \begin{bmatrix} 2 & 4 & 1 \\ -1 & 0 & 2 \\ 2 & -3 & -1 \end{bmatrix} \quad \text{and} \quad \mathbf{x} = \begin{bmatrix} -1 \\ 1 \\ 2 \end{bmatrix},$$

then

$$\mathbf{c} = \begin{bmatrix} 2 \cdot (-1) + 4 \cdot 1 + 1 \cdot 2 \\ (-1) \cdot (-1) + 0 \cdot 1 + 2 \cdot 2 \\ 2 \cdot (-1) + (-3) \cdot 1 + (-1) \cdot 2 \end{bmatrix} = \begin{bmatrix} 4 \\ 5 \\ -7 \end{bmatrix}.$$

The Octave instructions to compute the product $\mathbf{c} = A\mathbf{x}$ are

```
A=[2, 4, 1; -1, 0, 2; 2, -3, -1];
x=[-1; 1; 2];
c=A*x
```

Row Vectors via the Instruction " : "

a:step:b

A row vector may also be defined using the instruction `a:step:b` where `a` and `b` are real numbers and `step` represents the counting step.

The result of `a:step:b` is a row vector containing all values between `a` and `b` with step equal to `step`. The value of `step` can be omitted if it equals one.

For example the instruction

```
v=2:3:10
```

produces the row vector

```
v =
  2   5   8
```

and the instruction

```
v=0:.5:2
```

produces the row vector

```
v =
  0.00000   0.50000   1.00000   1.50000   2.00000
```

The instruction (without `step`)

```
v=1:10
```

gives the row vector

```
v =
  1   2   3   4   5   6   7   8   9   10
```

because it is implicitly assumed that `step=1`.

As a last example let us consider negative `step`. For the outcome vector to be non-empty, the value `a` must be larger than `b`. For example the instruction

```
v=10:-1:1
```

produces

```
v =
  10   9   8   7   6   5   4   3   2   1
```

while

```
v=1:-1:10
```

returns the empty vector

```
v = [ ]
```

since $a = 1 < b = 10$ (we should have $a > b$ when the step is negative).

Simple Vector Operations

The sum of the components of a vector v (either row or column) can be executed by the command sum. Given the row vector $\mathbf{v} = [2,\ 3,\ -2,\ 4,\ 0]$, to compute

$$s = v_1 + v_2 + v_3 + v_4 + v_5 = \sum_{i=1}^{5} v_i,$$

we type

sum(v)

```
v= [2, 3, -2, 4, 0];
s=sum(v)
```

giving

```
s = 7
```

The product of the components of a vector v (row or column) may be computed by the command prod. Given $\mathbf{v} = [2,\ 3,\ -2,\ 4,\ 2]$, to compute

prod(v)

$$p = v_1 \cdot v_2 \cdot v_3 \cdot v_4 \cdot v_5 = \prod_{i=1}^{5} v_i,$$

we type

```
v= [2, 3, -2, 4, 2];
p=prod(v)
```

and obtain

```
p = -96
```

The largest component of a (row or column) vector v is found by the command max(v).

max(v)

For instance, for $\mathbf{v} = [-5,\ 3,\ -2,\ 6,\ 2]$, the instructions

```
v= [-5, 3, -2, 6, 2];
M=max(v)
```

will return

```
M = 6
```

as the largest component.

Similarly, the smallest component of a (row or column) vector v is given by the command min(v). For instance

min(v)

```
v= [-5, 3, -2, 6, 2];
m=min(v)
```

produce

```
m = -5
```

abs(v)

Finally, the command abs(v) defines a new vector whose components are the absolute values of the components of v. Taking

```
v= [-3, 2, -1, 1, 0];
```

the command

```
w=abs(v)
```

produces

```
w= [3, 2, 1, 1, 0]
```

Augmenting Arrays (Adding Components to Vectors)

Row and column vectors can be modified by appending additional components and thus altering their dimension. Let us consider for instance the row vector

```
v = [2, 4, -1];
```

v=[v, 3]

to add a new entry (say, 3) to v, we type

```
v=[v, 3]
```

or

```
a=3;
v=[v, a]
```

and Octave will produce:

```
v =
   2   4  -1   3
```

If instead we wish to append the extra component 2 at the end of the column vector

```
w = [-1; 3; 0]
```

w=[w; 2]

we should type

```
w=[w; 2]
```

or

```
b=2;
w=[w; b]
```

and Octave will give:

```
w =
  -1
   3
   0
   2
```

With these instructions a vector can be augmented multiple times, but one should be careful to use commas and semicolons coherently with the type of vector. For instance if v is a row vector, the instruction v=[v; 3] will give an error, just as w=[w, 3] will give error if w is a column vector.

"Element-by-Element" Operations

Element-by-element operations are: `.^` (power), `.*` (product), and `./` (division) and they are performed on vectors or matrices, term by term. We explain how to use them in the following exercise.

Exercise: Element-by-element operations on a vector

Given the row vector $\mathbf{x} = [8, -4, 5]$, construct the row vector $\mathbf{y}$ with components $y_i = x_i^2 + 1$ (with $i = 1, \ldots, 3$) and the row vector $\mathbf{z}$ with components $z_i = 1/x_i$ (with $i = 1, \ldots, 3$). Then compute the row vectors $\mathbf{v}$ of components $v_i = x_i/y_i$, and $\mathbf{w}$ of components $w_i = x_i y_i$ (for $i = 1, \ldots, 3$).

Solution. Each component of $\mathbf{y}$ is the square of the corresponding component of $\mathbf{x}$ plus 1. To obtain $\mathbf{y}$ we must use the operation with symbol `.^`("point power"). The instructions:

`.^`

```
x=[8, -4, 5];
y=x.^2+1
```

produce

```
y =
  65  17  26
```

Beware that the instruction

```
y=x^2
```

is wrong, and Octave will give the following error message:

```
error: for x^A, A must be a square matrix.
Use .^ for elementwise power.
```

To define the vector $\mathbf{z}$ we use the operation `./` ("point divided") as follows

`./`

```
z=1./x
z =
  0.12500  -0.25000  0.20000
```

The command to get $\mathbf{v}$ is

```
v=x./y
```

which gives

```
v =
  0.12308 -0.23529  0.19231
```

whilst for $\mathbf{w}$ we use the operation `.*` ("point star") as follows

`.*`

```
w=x.*y
```

The result is

```
w =
  520  -68  130
```

Exercise: Computation and plot of functions

We want to compute the function $f(x) = x^4 - 3x^3 - 2x^2 + 4x + 1$ at 101 evenly spaced points in the interval $I = [-1, 1]$ and then plot it.

Solution. Take $n \geq 2$. The relations:

$$x_1 = a, \ x_2 = x_1 + h, \ x_3 = x_2 + h, \ \ldots, \ x_n = x_{n-1} + h = b,$$

where $h = (b - a)/(n - 1)$, define n equally spaced points in the interval $[a, b]$, with endpoints included.

Let us build a vector $\mathbf{x}$ of dimension 101 whose ith component is the point x_i defined above. The Octave command is

linspace

```
x=linspace(-1, 1, 101);
```

More generally, the instruction

```
x=linspace(a, b, n);
```

generates a row vector whose components are n evenly spaced values between a and b.

Now we want to construct a vector $\mathbf{y}$ with components $y_i = f(x_i)$, the images of the points x_i under the function f given by the problem. To this end let us define the function handle of f by the command

```
f=@(x)x.^4-3*x.^3-2*x.^2+4*x+1;
```

observe we have used element-by-element operations to compute f on all of the components of `x` using a single command. The command we need for storing in $\mathbf{y}$ the values of f at all nodes x_i $(i = 1, \ldots, 101)$ is

```
y=f(x);
```

The components `x(1)` and `y(1)` of `x` and `y` contain the coordinates of the point $P_1 = (x_1, y_1)$; the components `x(2)`, `y(2)` contain the coordinates of the point $P_2 = (x_2, y_2)$ and so forth, up to $n = 101$. By the `plot(x, y)` command we plot a polygonal line[4] joining $P_1, P_2, P_3, \ldots$, up to P_{101}:

plot

```
figure(1); clf
plot(x, y)
grid on
xlabel('x')
ylabel('f(x)')
```

and the output is shown in Fig. 3.3. □

The general syntax of the command `plot` is:

```
plot(x, y, linespec)
```

[4] A *polygonal line* is the union of consecutive straight segments.

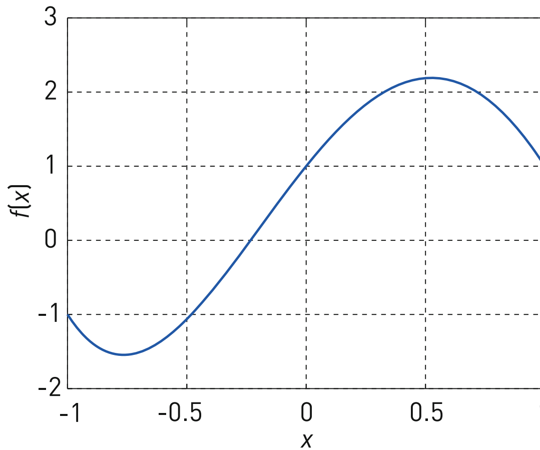


Fig. 3.3 The Octave output of the commands of problem *Computation and plot of functions*

where `linespec` is a string that specifies the colour, the type of line and a marker. For example

```
plot(x, y, 'r--')
```

draws a red (`r`=red) dashed line,

```
plot(x, y, '-*')
```

draws a solid line in the default colour with star-type marker, while

```
plot(x, y, 'g')
```

simply draws a green (`g`=green) solid line. For a detailed description of all possible options the reader should consult the `help` of the `plot` command.

In case of a large number of points (as 101, above), the polygonal line drawn by the `plot` command looks rather smooth and provides a good approximation of the actual graph of the function f.

Try for example to redo exercise *Computation and plot of functions* with 9 points instead of 101, and you will see that the final graph is a polygonal line that differs from the graph of $f(x) = x^4 - 3x^3 - 2x^2 + 4x + 1$. In order to plot a function accurately it is always a good idea to employ many points.

The `plot` command is an alternative to `fplot`, seen earlier. The difference between the two is essentially that with `plot` we choose the points at which f is evaluated (the nodes of the polygonal line), whereas with `fplot` the choice is made automatically by Octave.

Printing Instructions

To print on screen strings of alphanumerical characters and the results of operations, we can use the commands `disp` and `fprintf`.

disp

The disp command only allows to print on screen strings of characters. For example, the instruction

```
disp('Hey there, I''m using Octave')
```

displays

```
Hey there, I'm using Octave
```

Anything we want to screen-print must be enclosed by round brackets.

Let us emphasise the double quote in I"m: whenever we want to print a single-quote mark, it must be typed twice, because normally Octave interprets the single quote as the end of the string to be printed.

fprintf

The command fprintf permits to print on screen strings of characters and variables on the same row, using special formats. For example the instructions

```
n=3; A=[2, sqrt(2), pi];
fprintf('The component %d of A contains the value %f \n', n, A(n))
```

will display

```
The component 3 of A contains the value 3.141593
```

The command fprintf screen-prints the string within quotes, where: the format %d is substituted by the content of the variable n written after the string, while the format %f is substituted by the content of the variable A(n) appearing after n.

During the printing phase each format character (%d and %f in the example) is substituted by the content of a variable. After the string, therefore, we must write as many variables as the number of format characters used in the string (two, in our case). The variables will be associated with the formats according to the order in which they are written.

The formats used by Octave for screen display are shown in Table 3.2.

The last two characters \n at the end of the string in the fprintf command tell Octave that the line has ended, and they force it to move on to a new line.

When specifying a format we may also decide the number of characters to be used to print a certain variable. Referring to the previous printing instruction, for example, if we want to allow a total of 16 characters for A(n), of which 14 for the decimal part, we can use the format %16.14f instead of %f. The instructions

```
n=3; A=[2, sqrt(2), pi];
fprintf('The component %d of A contains the value %16.14f \n', ...
     n, A(n))
```

Table 3.2 Octave formats for screen display

Variable type	Format
integer	% d
real (fixed format)	% f
real (exponential format)	% e
character string	% s

display

```
The component 3 of A contains the value 3.14159265358979
```

Note how we may write several Octave instructions on the same line.

3.1.4 Programming Units: Script and User-Defined Functions

The Script: A Shopping List of Sorts

A *script* is a file with extension `.m` that contains one or more Octave instructions.

A script is a bit like a shopping list where we jot down what we need to buy. Instead of writing the instructions one at a time in the command window (as if we were purchasing one item at a time), we write them all in one file (the script) and then execute the script using a single command (we buy everything in one go).

Let us examine the steps we need for writing and executing a script.

1. We select “File” in the Octave bar and then “New script”; this moves us to the Editor window where there is no prompt »).
2. In this window we write the Octave instructions. For example to plot the functions $f(x) = x^2$ and $g(x) = x^3$ over the interval $[-1.5, 1.5]$ type:

   ```
   f=@(x)x.^2; g=@(x)x.^3;
   x=linspace(-1.5, 1.5, 200); yf=f(x); yg=g(x);
   figure(1); clf
   plot(x, yf); hold on; plot(x, yg);
   xlabel('x'); ylabel('y');
   legend('f(x)=x^2', 'g(x)=x^3')
   ```

3. We then save the file by selecting “File” and “Save as” in the Editor menu (or clicking on the ‘save’ icon), and type the name `firstscript` (Octave automatically assigns the extension `.m` to the file).
4. We return to the command window, the one with the prompt, and type

   ```
   firstscript
   ```

   A graphical window will pop up (containing the plots of the functions f, g), like that in Fig. 3.4.

Should Octave flag errors, we must go back to the editor window, modify the script and amend the errors, then save, return to the command window and re-type the command `firstscript`.

If Octave says it cannot find the command `firstscript`, it means we have saved the script in a folder Octave does not usually read. So we must tell Octave to go and find the script, by typing (in the command window before calling up the script) the command

addpath

```
addpath directory
```

`directory` can be an absolute or relative *path*, and it abides by the standard rules for accessing files through the tree-like structure of the file system.

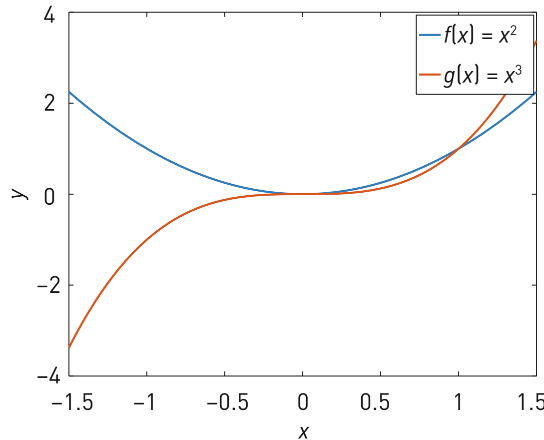


Fig. 3.4 The plot of functions f and g

For example, if we are working in Windows and we have saved the file in the folder `C:/Users/John/Octave`, the command

```
addpath C:/Users/John/Octave
```

tells Octave to search for scripts in this folder, too.

If we now wish to display the graph of $g(x) = x^3 + 1$ instead of $g(x) = x^3$, it is enough to modify the first line in the script, save, and then just type `firstscript` in the command window, without having to rewrite the commands all over again.

The variables defined in a script have a global value, meaning they remain accessible also after the execution of the script. What is more, inside the script we may reference (read or modify) variables that were defined before launching the script.

User-Defined Functions or, for Short, Functions

A *user-defined function* is a function written by the user, that is to say, us. It is a true programming unit containing Octave instructions and it has input and output variables.

To define a user-defined function (from now on simply called a *function*), we should open an editor window (as we did to generate a script), insert as first line of the function an instruction of the sort

```
function [out1, out2, .., outn]=myfunction(in1,in2, .., inm)
```

and then all the instructions that constitute the function.

`out1, .., outn` are the *output variables*, i.e. the *outcome* of the function (they are always enclosed by square brackets and separated by a comma), while `in1, .., inm` are the *input variables*, the *data* used by the function to perform the computations (always enclosed in round brackets and separated by commas).

Next we have to save the function in the file `myfunction.m`. To execute it we must write in the command window:

```
[out1, out2, .., outn]=myfunction(in1, in2, .., inm)
```

Let us see an example of how to write a function and have Octave execute it.

Exercise: Our first function

Write a function, called `plot_f`, that takes a mathematical expression $y = f(x)$ and the endpoints a, b of an interval as input, evaluates the function at 500 evenly spaced points in $[a, b]$, plots the graph and returns the value of f at the midpoint of the interval.

Solution. In the horizontal menu below the command window we click on “Editor” and open a new script (either using the icon or selecting “File”, “New script” in the menu). In the editor window write these commands:

```
function [ymp]=plot_f(f, a, b)
% plot_f: plots a mathematical function f on the interval [a, b]
% and computes f at the midpoint of [a, b]
% Calling instruction: [ymp]=plot_f(f, a, b)
% Input: f = function handle of the function to be plotted
%     a = left endpoint of the interval
%     b = right endpoint of the interval
% Output: ymp = value of f at the midpoint of the interval
x=linspace(a, b, 500); % generates a vector of 500 components
y=f(x);  % computes the y-values
figure(1); clf % selects the graphical window and wipes it
plot(x, y)  % plots the graph of f
xlabel('x'); ylabel('y=f(x)') % defines labels for the axes
xmp=(a+b)/2; % computes the midpoint of the interval
ymp=f(xmp); % evaluates f at the midpoint
```

Then we save the function under `plot_f` (exactly the same name written in the first line of the function) and go back to the command window.

Before executing the `plot_f` function we must define the input variables `f`, `a` and `b`. For example set

```
f=@(x)x.^3-2*x+1; % defines a function handle as we intend
a=-1; b=1.5; % define the endpoints of the interval
```

Now we execute the function by typing the command

```
[ymp]=plot_f(f, a, b)
```

Then, the value

```
ymp = 0.51562
```

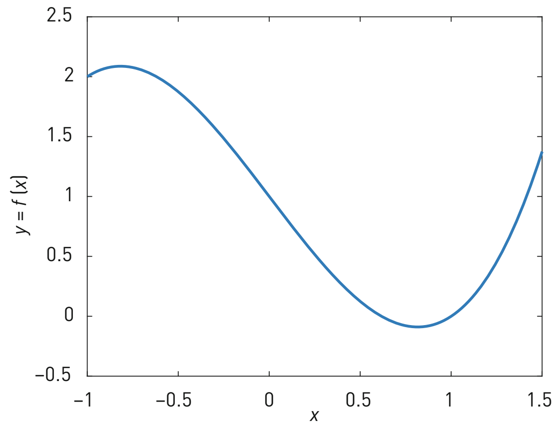


Fig. 3.5 The output plot of the `plot_f` function

will appear on the screen and a graphical window will show up with the graph of $f(x) = x^3 - 2x + 1$, see Fig. 3.5.

If we type

```
help plot_f
```

Octave says:

```
% plot_f: plots a mathematical function f on the interval [a, b]
% and computes f at the midpoint of [a, b]
% Calling instruction: [ymp]=plot_f(f, a, b)
% Input: f = function handle of the function to be plotted
%     a = left endpoint of the interval
%     b = right endpoint of the interval
% Output: ymp = value of f at the midpoint of the interval
```

which is an onscreen display of what we wrote in the commented rows at the beginning of the function; this is displayed in the command window and serves as the `help` of the command itself. Our function has turned into an Octave command.

Suppose we wanted to use the function `plot_f` to draw another mathematical function. There is no need to change the definition of `plot_f` because it suffices to redefine the function handle `f` before we call the `plot_f` function. For example to plot $f(x) = x^3 - 5x$ on $[-0.5, 1.75]$ and compute f at the midpoint it suffices to type the commands

```
f=@(x)x.^3-5*x; a=-0.5; b=1.75; [ymp]=plot_f(f, a, b)
```

□

The variables used inside the function that are not output variables (like `x`, `y`) are lost as soon as the function has done its job.

Variables defined before the execution of the function can be used inside the function only if they are listed as input variables.

3.1.5 Fundamental Statements

The for Loop

We introduced `for loops` in Chap. 2, Sect. 2.2. Now we show how to implement them in Octave. Consider a row vector `v` (say, `v=[1, 2, 3]`). The structure of a `for loop` is:

```
for k=v
  <instructions>
end
```

The words `for` and `end` are *keywords* in Octave[5] and they delimit the *for loop*. The for loop repeats the instructions written inside of it for every value of the counter `k` contained in the row vector `v`.

The instructions

for

```
for k=[1 2 4]
  fprintf('print the number %d \n', k)
end
```

for instance, produce this output

```
print the number 1
print the number 2
print the number 4
```

The vector `v` used in the `for` construct may be replaced by an instruction like `a:step:b` as we have seen on p. 64.

When the variables `k1` and `k2` contain integers with `k1<k2`, the loop

```
for k=k1:k2
  <instructions>
end
```

will perform the instructions specified between the beginning and the end of the loop for all values of `k` between `k1` and `k2`.

Exercise: Computing a sum

Compute the sum of the reciprocals of the first 10 natural numbers, i.e. the real number

$$s = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{9} + \frac{1}{10} = \sum_{k=1}^{10} \frac{1}{k}.$$

[5]The keywords of the language cannot be used as variable names. To know whether a certain word is a keyword, type the command `iskeyword('name')`. The answer will be 1 (= true) if the word `name` is a keyword in Octave, otherwise it will be 0 (= false).

Solution. The symbol $\sum_{k=1}^{10} \frac{1}{k}$ denotes the sum of terms $\frac{1}{k}$, one for each value of k between the first, $k = 1$, and the last, $k = 10$.

Let us use the variable `s` to store the sum. We initialise the sum at value zero and add to it one term at a time. The Octave instructions read:

```
s=0;
for k=1:10
  s=s+1/k;
end
s
```

and we obtain the value `s = 2.9290`. □

Loops may be nested, as the next exercise clarifies.

Exercise: Assemble the Hilbert matrix

Build the matrix A with elements

$$a_{ij} = \frac{1}{i + j - 1} \qquad i = 1, \ldots, 5 \text{ and } j = 1, \ldots, 5.$$

Such a matrix is known as a *Hilbert matrix*.

Solution. Even if Octave does not require that we declare variables and fix their size (as opposed to other programming languages like C, C++ and Fortran), it is highly recommended to initialise matrices and vectors, especially when their size is large. The command to initialise a matrix of n rows and m columns with zero entries everywhere is `zeros(n,m)`.

zeros

Let us define A by the Octave instructions:

```
n=5; % variable n containing the dimension of A
A=zeros(n, n);% initialises an n x n matrix with all zero entries
for i=1:n  % loop over the row index i
 for j=1:n   % loop over the column index j
   A(i, j)=1/(i+j-1);  % assigning instruction
 end       % end of loop over j
end        % end of loop over i
A          % print A on the screen
```

(Note that these are two nested loops.) The output is:

```
A =
   1.00000   0.50000   0.33333   0.25000   0.20000
   0.50000   0.33333   0.25000   0.20000   0.16667
   0.33333   0.25000   0.20000   0.16667   0.14286
   0.25000   0.20000   0.16667   0.14286   0.12500
   0.20000   0.16667   0.14286   0.12500   0.11111
```

By typing

```
format rat; A
```

we can display the matrix elements as fractions (rational numbers). By printing on screen the content of `A` (just type `A` in the command window of Octave), we obtain

```
A =
 1       1/2     1/3     1/4     1/5
 1/2     1/3     1/4     1/5     1/6
 1/3     1/4     1/5     1/6     1/7
 1/4     1/5     1/6     1/7     1/8
 1/5     1/6     1/7     1/8     1/9
```

In order to return to the standard Octave format we type the command

```
format short
```

Now that we have introduced `for loops`, let us see how we can generate an empty vector and then fill it with values. By the instructions:

`v=[ ]`

```
v=[ ]; % initialises the empty vector v
w=[ ]; % initialises the empty vector w
for k=1:5
 v=[v, 1/k];
 w=[w; -k];
end
v       % print the vector v on the screen
w       % print the vector w on the screen
```

we generate a row vector `v` (we have used commas to add new elements) containing the reciprocals of the first five positive integers, and a column vector `w` (we have used semicolons to include the new components, so they are added on different rows, hence as a column) containing the opposites of the first five positive integers. At the end of the loop the vectors read:

```
v =
   1.00000   0.50000   0.33333   0.25000   0.20000
```

and

```
w =
 -1
 -2
 -3
 -4
 -5
```

Selection Blocks

The simplest Octave statements for making choices are described below.

1. *Condition without alternative*

```
if <condition>
 <instructions>
end
```

If the condition written in `<condition>` is satisfied, the instructions in the block `<instructions>` are executed. For example, the instructions for

implementing the control: *"if the variable* `n` *exists (meaning: it has been initialised), then double its content"*, are

```
if exist('n')
 n=n*2
end
```

where the command `exist('n')` gives back the logical value 'true' if the variable `n` exists (it was initialised by a numerical value) and 'false' otherwise. If we type

`exist`

```
n=3;
if exist('n'), n=n*2, end
```

Octave will say `n=6`, whilst with

```
clear n
if exist('n'), n=n*2, end
```

Octave will just show the prompt without writing anything (remember Octave allows to write several instructions on the same command line).

2. *Condition with one alternative*

```
if <condition>
 <instructions 1>
else
 <instructions 2>
end
```

If the condition in `<condition>` is satisfied, the instructions in the `<instructions 1>` block are executed; if not, the instructions in block `<instructions 2>` are executed. For example, the instructions to execute the control: *"if the variable* `n` *exists, double its content; if not, assign to it the value 1"*, are

```
if exist('n')
 n=n*2;
else
 n=1;
end
```

3. *Condition with many alternatives*

```
if <condition 1>
 <instructions 1>
elseif <condition 2>
 <instructions 2>
elseif <condition 3>
 <instructions 3>
...
else
 <instructions n>
end
```

If the condition in `<condition 1>` holds, the instructions in block `<instructions 1>` are executed; otherwise, if the condition in `<condition 2>` holds, the instructions in `<instructions 2>` are executed and so on,

until we reach the last alternative, expressed by `else`. We will provide an example for this case after introducing relational operators.

Also `if`, `else` and `elseif` are keywords in Octave.

Relational Operators

A condition must always return the truth value *true* or *false*. In Octave, true and false are respectively represented by the numerical values 1 and 0. Often a condition is built using *relational operators*, which are the operators comparing numbers and alphanumerical strings using an order relation. Relational operators in Octave are codified as shown in Table 3.3. Examples of conditions built using relational operators include: $x \neq 0$, $a > b$ and $n \leq 3$.

The result of a comparison obtained by a relational operator is a variable of logical or Boolean type, one that can only assume truth value 1 (true) or 0 (false).

For example the instructions:

```
a=3; b=4;
b>a
```

give as output

```
ans = 1
```

i.e. the value 'true' (since 4 is larger than 3).

Now we are ready for the promised example of a condition with many alternatives. Assuming the variable `n` has been already initialised and contains a numerical value (we leave it to you to give `n` the value you prefer), we write the Octave instructions implementing the following control: *"if the variable* `n` *contains a negative value, make it the opposite; otherwise, if it contains a positive value, multiply it by two; otherwise add one to it."* So here you go:

```
if n <0
 n=-n
elseif n>0
 n=n*2
else
 n=n+1
end
```

Table 3.3 Relational operators in Octave

Relational operator	Octave code
bigger $>$	`>`
bigger or equal $\geq$	`>=`
smaller $<$	`<`
smaller or equal $\leq$	`<=`
equal[6] $=$	`==`
different $\neq$	`~=` or `!=`

[5]The Octave symbol for the equality relational operator is ==. The symbol = is used for assignments.

Let us now see how to compute the factorial of a number in Octave.

Exercise: Computing the factorial of a number

The factorial of a natural number $n \in \mathbb{N}$ is

$$n! = 1 \cdot 2 \cdot 3 \cdots (n-1) \cdot n = \prod_{k=1}^{n} k \quad \text{if } n > 0,$$

and $0! = 1$. Write an Octave function that returns the factorial of an input $n \in \mathbb{N}$.

Solution. Recall that $\prod_{k=1}^{n} k$ means we must multiply all numbers k between the values below and above the symbol $\prod$, i.e. 1 and n.

The function we need is:

```
function [f]=myfactorial(n)
% myfactorial: computes the factorial of a natural number
% Calling instruction: f = myfactorial(n)
% Input: n = natural number (n >= 0)
% Output: f = n!
if n<0
  disp('you wrote n<0, n! isn''t defined')
  f=[];
elseif n==0
  f=1;     % f contains 0! = 1
elseif n==1
  f=1;     % f contains 1! = 1
else
  v=2:n;   % v is a row vector containing the values 2,3,...,n
  f=prod(v); % computes the product of the components of v
end
```

The instruction

```
f=myfactorial(4);
```

produces

```
f = 24
```

The instruction

```
f=myfactorial(0);
```

returns

```
f = 1
```

while the instruction

```
f=myfactorial(-5);
```

gives

```
you wrote n<0, n! isn't defined
```

Octave requires we always define output variables, even if the conditions to evaluate them do not hold. In such a case it suffices to initialise them as empty variables, like `f=[ ]`.

In the above function we did just that for the case n negative, when we cannot compute the factorial. □

The selection block written for $n!$ produces the same result when $n = 0$ and $n = 1$. Therefore we could combine the results of the logical operations `n==0` and `n==1` using the *logical operator or*.

Logical Operators

Logical operators are operators acting on logical variables that produce another logical variable. Table 3.4 lists logical operators and their codes.

The *short-circuit* form of logical operators is useful to minimise the number of controls. For example, the instruction

```
n==0 | n==1
```

tells Octave to compute the truth value of both `n==0` and `n==1`, combine them under the logical operator 'or' and then return the result. With the instruction

```
n==0 || n==1
```

Octave computes just the truth value of the first expression `n==0`, and only in case the latter is false, it proceeds to compute the second one.

So now we can modify the selection block written in the `factorial` function, as follows:

```
if n<0
f=[]; disp('You wrote n<0. n! isn''t defined');
elseif n==0 || n==1
 f=1;
else
 v=2:n; f=prod(v);
end
```

While Loops

The syntax of `while loops` is the following:

`while`

```
while <condition>
 <instructions>
end
```

Table 3.4 Logical operators in Octave.

Logical operator	Octave code
not	˜ or !
and	&
(inclusive) or	|
short-circuit and	&&
short-circuit (inclusive) or	||

The instructions in the `<instructions>` block are repeated provided the condition in `<condition>` holds.

For example in:

```
n=3;
while n<=20
  n=n+1
end
```

the variable n is repeatedly increased by one as long as it stays less than or equal to 20. The moment n reaches 21, the addition stops and we exit the loop. At the end of the loop n contains the value 21.

Note that to avoid the `while loop` to go on indefinitely, it is necessary to modify the variable involved (or variables involved, if more than one) in the `<condition>` control inside the `<instructions>` block.

Exercise: Computer precision assessment

Compute the smallest positive number ϵ_M (among those that Octave can handle) whose sum with 1 is larger than 1, i.e. such that

$$1 + \epsilon_M > 1.$$

This ϵ_M is called *machine epsilon* or *computer precision*, and depends on how the machine stores the numbers on which it operates.

Solution. The idea for computing ϵ_M is to start from a variable $u = 1$ and halve it repeatedly as long as $1 + u \neq 1$. The first value of $u > 0$ for which the condition "$1 + u \neq 1$ " fails (the value for which $1 + u = 1$) will make us exit the `while loop`. Once we have such value u, the computer precision ϵ_M is given by $\epsilon_M = 2u$. Let us write a script with the following instructions

```
u=1;
while 1 + u~= 1
  u=u/2;
end
u        % u contains the largest positive value that
         % does not fulfil condition 1 + u ~= 1
epsilonM=u*2 % multiplying by 2 gives the required number
```

and save them in the file `precision`. Back in the command window, type

```
precision
```

to get

```
u = 1.1102e-16
epsilonM = 2.2204e-16
```

First of all let us interpret the output:

Exponential form of numbers

`1.1102e-16` should be read $1.1102 \cdot 10^{-16}$
`2.2204e-16` should be read $2.2204 \cdot 10^{-16}$

meaning that Octave has represented numbers in *exponential form*, and the letter `e` precedes the exponent, in base 10. So the answer to our problem is that the computer precision is

$$\epsilon_M \simeq 2.2204 \cdot 10^{-16}.$$

In other words the smallest positive number, among those handled by Octave, that added to 1 is different from 1 is $\epsilon_M \simeq 2.2204 \cdot 10^{-16}$. □

Let us go over what we have just done. To exit the while loop Octave has determined that $u = 1.1102 \cdot 10^{-16}$ (a tiny number, but non-zero) is such that $1 + u = 1$!!

This is rather surprising since we all know very well that inside the reals $\mathbb{R}$ zero is the only number that added to any number x equals x itself.

When we work on a computer this crucial arithmetic property no longer holds (we shall deal with this in Sect. 3.2 of the current chapter).

If we are still shocked, let us do another check. Typing

```
u
```

gives

```
u = 1.1102e-16
```

while by

```
1+u
```

we get

```
ans = 1
```

so $1 + u$ equals 1 on the computer!

3.1.6 Tidbits of Graphics

In this subsection we will explain some instructions for more elaborate plots than the ones seen so far, where we will be able to prescribe the width of lines, the size of symbols, or the text font. We shall also see how to prepare the graphs needed in subsequent chapters.

Let us consider the functions $f(x) = \dfrac{|x-3|}{x^2+4}$ and $g(x) = \dfrac{4+3x-x^2}{8}$ over the interval $[-1, 4]$ and plot them by:

```
x=linspace(-1,4,50); % generates the nodes x
f=@(x)abs(x-3)./(x.^2+4); % defines the function handle for f
g=@(x)(4+3*x-x.^2)/8; % defines the function handle for g
yf=f(x); yg=g(x); % evaluates f and g
figure(1); clf % selects figure 1 and wipes it
plot(x,yf,'Linewidth',2) % plots the function f
hold on; grid on; % keeps the graph and draws the grid
plot(x,yg,'.','Markersize',16) % plots g
xlabel('x','Fontsize',20); % assigns a label to the x-axis
ylabel('y','Fontsize',20) % assigns a label to the y-axis
le=legend('f(x)','g(x)'); % defines the caption
set(le,'Fontsize',20,'Location','Northwest'); % specifies
                       % properties of the caption
set(gca,'Fontsize',20) % specifies properties of the graph
```

We will just describe the instructions (or specifications) that we have not met yet. With the instruction

```
plot(x, yg, '.', 'Markersize', 16)
```

we are telling Octave to plot the graph with dots of size 16 pt (pt means *typographic point* and is a unit of measure in typography).

With

```
xlabel('x','Fontsize',20);
ylabel('y','Fontsize',20);
```

we define the labels for the axes by specifying that the font must be 20 pt. The instruction

```
le=legend('f(x)','g(x)');
```

defines the legend and saves its identifier in the variable `le` (`le` is the name we associate with the legend of the graph we are working on). If we want to specify or change the properties of the legend, we should use the command `set` and refer to the `le` identifier. For example, the instruction

`set`

```
set(le,'Fontsize',20,'Location','Northwest');
```

tells Octave that the object identified by `le` (the legend) must have a 20 pt font and its position within the graph must be at the top left (Northwest is like top left on a geographical map).

Finally, the instruction

```
set(gca,'Fontsize',20)
```

tells Octave that the object identified by `gca` has a 20 pt font. `gca` is an Octave variable that identifies the *current axes*, i.e. the graph currently in use.

`gca`

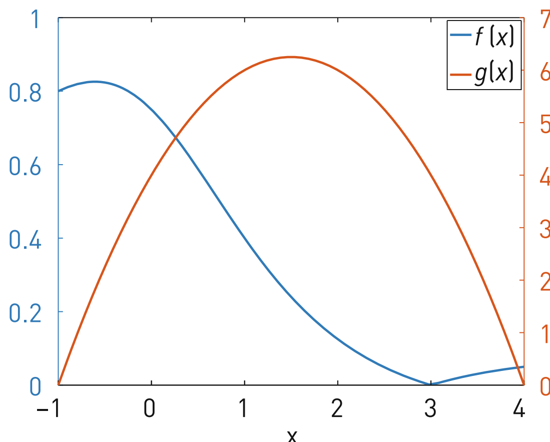


Fig. 3.6 The output of command `plotyy`

Different Scales on the y-Axis in the Same Graph

plotyy

The command `plotyy` allows us to draw two functions (or the content of two vectors) using different scales on the y-axis. For example, the instructions

```
x=linspace(-1,4,50);
f=@(x)abs(x-3)./(x.^2+4); g=@(x)4+3*x-x.^2;
yf=f(x); yg=g(x);
figure(1); clf
[ax,h(1),h(2)]=plotyy(x,yf,x,yg); % plots two overlapping graphs
% with different scaling, one on the left and one on the right
set(h(1),'Linewidth',2);% modifies the linewidth of f
set(h(2),'Linewidth',2);% modifies the linewidth of g
set(ax(1),'Fontsize',20);% modifies the font on the first graph
set(ax(2),'Fontsize',20);% modifies the font on the second graph
le=legend('f(x)','g(x)'); set(le,'Fontsize',20)
xlabel('x','Fontsize',20)
```

produce the graph in Fig. 3.6 where the vertical (y) scaling for f is on the left, and the scaling for g is on the right. The colours used for the graphs match the colours of the scalings. The output of `plotyy` are: `ax` is the vector containing the handles of the two graphs (in reality the command `plotyy` has drawn two distinct but overlaying graphs), `h(1)` is the handle of the axis generated by the plot of f, and `h(2)` the handle of the axis generated by the plot of g. Font size and line width are modified by the command `set`.

Graphs in Logarithmic Scale

To obtain graphs in logarithmic scale (see Chap. 6, Sect. 6.4.5 for the description of the logarithmic scale) we must use the command `loglog(x, y)`, where `x` and `y` are the vectors containing the (positive) abscissae and (positive) ordinates of the data we want to represent.

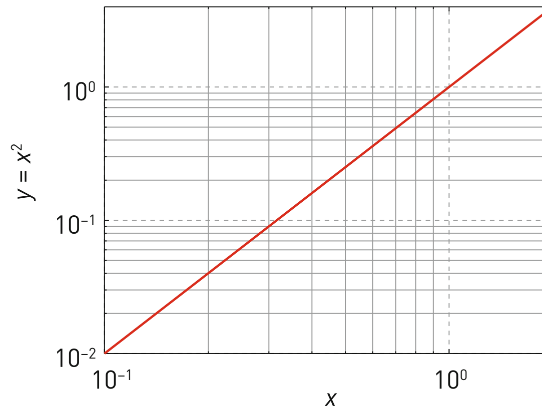


Fig. 3.7 The function $y = x^2$ in logarithmic scale, with x in $[10^{-1}, 2]$

loglog

For example:

```
x=linspace(0.1,2,100); % generates the vector of abscissae
y=x.^2;          % generates the vector of ordinates
figure(1); clf
loglog(x,y,'r')      % representation in log scale
grid on
xlabel('x'); ylabel('y=x^2')
axis([1.e-1,2,1.e-2,4]) % redefines the boundaries of the graph
```

produce Fig. 3.7.

The `axis` command defines the endpoints of the intervals (horizontal and vertical) that can be represented in the graph. In order to tell Octave to plot data with x between x_{min} and x_{max} and y between y_{min} and y_{max} we should give the instruction

axis

```
axis([xmin, xmax, ymin, ymax])
```

Exercises

Solve Exercise 3.1, p. 211, Exercise 3.2, p. 214, and Exercise 3.3, p. 215.

3.2 How Computers Represent Numbers

The set of real numbers, denoted by $\mathbb{R}$, is the set we are accustomed working with when we make calculations by hand. In $\mathbb{R}$ we can do algebraic operations, extract roots, compute logarithms, evaluate trigonometric functions and compute limits, derivatives, integrals etc.

We know it has infinite cardinality, and while the rationals possess a finite or periodic decimal expansion, irrational numbers have infinitely many decimal digits that do not repeat periodically.

In a Tight Spot

The first hurdle that we meet when working with computers is the storage of numbers. Numbers are encoded in registers made of a finite number of bytes, usually 4 or 8, and it is not possible to store a real number with infinitely many decimal digits in a register of finite length. Therefore not every real number will be faithfully representable by a computer.

To store and represent numbers, Octave uses the exponential notation. This allows to work with both very small and very large numbers, such as $x = 10^{-15}$ and $c = 1.001 \cdot 10^{308}$, which in Octave would be written

```
x=1e-15
c=1.001e308
```

The letter `e` stands for "times 10 to the power", and separates the decimal part of the number (before the `e`) from the exponent in base 10 (after the `e`). If the exponent is positive its sign can be omitted, but when it is negative the minus sign must be placed after the letter `e`.

Let us start from a very easy example that will make us understand the problems that we are facing, but cannot avoid, when we compute using a machine (keep in mind that the calculator on your phone has the same problem).

An Elementary, But Inevitable, Mistake

Let us consider the following operation

$$\frac{(-1 + (1 + x))}{x}$$

defined for any $x \neq 0$. Its exact result is clearly 1 for any $x \neq 0$. Choosing for instance $x = 12.345$ and writing the following instructions:

```
x=12.345;
y=(-1+(1+x))/x
```

Octave produces the correct result:

```
y = 1
```

Now take $x = 10^{-15}$: the instructions

```
x=1e-15;
y=(-1+(1+x))/x
```

will produce

```
y = 1.1102
```

which is quite different from 1! The relative error (relative to the exact value 1) made by the machine in computing `y` is

$$\frac{|1 - \mathtt{y}|}{1} = |1 - 1.1102| = 0.1102,$$

which amounts to an 11% discrepancy. So let us try to modify the order of the operations in the numerator (shuffling the inner brackets): instead of $(-1 +$

$(1 + x))/x$ we compute $((-1 + 1) + x)/x$ (obviously the two expressions are mathematically the same). The instructions

```
x=1e-15;
y=((-1+1)+x)/x
```

produce the correct result

```
y = 1
```

In computer arithmetic, it is not always the case that $(-1 + (1 + x))$ equals $((-1 + 1) + x)$ (or more generally, that $(a + (b + c))$ equals $((a + b) + c)$).

This implies that there exist numbers $x \neq 0$ for which

$$\frac{(-1 + (1 + x))}{x} \neq \frac{((-1 + 1) + x)}{x}$$

on a computer. This simple example should warn us that the outcome of computer operations may differ a lot from those done by hand. As we shall see better in a short while, certain standard arithmetic properties (in this case, the associative property of addition[6]) are violated in computer arithmetic. Let us try to understand how a computer represents numbers and how it handles them.

3.2.1 Floating-Point Numbers

When we employ a computer for calculations we must content ourselves with using so-called *floating-point numbers*, which are approximations of real numbers.

The set of floating-point numbers is denoted by the letter $\mathbb{F}$ and its elements have the following exponential representation

$$x = (-1)^s \cdot (0.a_1 a_2 \ldots a_t) \cdot \beta^e = (-1)^s \cdot m \cdot \beta^{e-t}, \quad a_1 \neq 0 \tag{3.1}$$

where:

- s is 0 or 1, so $(-1)^s$ defines the sign of the number;
- β (a positive integer bigger than or equal to 2) is the *base* used to represent numbers on the computer;
- $m = a_1 a_2 \cdots a_t$ is an integer called the *mantissa*;
- $a_1, a_2, \ldots, a_t$ are the digits of the mantissa, natural numbers between 0 and $\beta - 1$, constrained by $a_1 \neq 0$;
- t is the maximum number of mantissa digits that can be stored in a register inside the memory of the computer;
- the number e is called *exponent*, and can range between $L < 0$ and $U > 0$.

The set $\mathbb{F}$ is therefore characterised by the base β, the number t of significant digits in the mantissa and by the extreme values L, U of the exponent e. It is denoted by

$$\mathbb{F}(\beta, t, L, U) \tag{3.2}$$

[6]We remind that in exact arithmetic $(a + b) + c = a + (b+c)$, for all real numbers a, b, c.

An example of $\mathbb{F}$ set

The set $\mathbb{F}(10, 5, -3, 4)$ has $\beta = 10$, $t = 5$, $L = -3$ and $U = 4$, so it contains all numbers that can be written in form (3.1) with: base $\beta = 10$, 5-digit mantissa, exponent e between -3 and 4.

Therefore the numbers $x = 0.12345 \cdot 10^{-3}$, $y = 0.23821 \cdot 10^{-1}$, $z = -0.87431 \cdot 10^2$ belong to $\mathbb{F}(10, 5, -3, 4)$.

The mantissa of x is $m = 12345$, its exponent $e = -3$, and since $x > 0$ we have $s = 0$.

As for y we have $s = 0$, $m = 23821$ and $e = -1$, while for z we have $s = 1$ (giving the minus sign), $m = 87431$, $e = 2$.

The numbers $w = 0.123456 \cdot 10^{-3}$, $v = -0.23821 \cdot 10^5$, $q = 0.87431 \cdot 10^{-5}$, instead, do not belong in $\mathbb{F}(10, 5, -3, 4)$, because w has a mantissa of length 6, not 5, v has exponent $e = 5 > U$, and q has exponent $e = -5 < L$.

The requirement $a_1 \neq 0$ prevents a number to have multiple representations. Without that condition, for example, the base-ten representation of $\frac{1}{10}$ could be $0.1 \cdot 10^0$, or $1 \cdot 10^{-1}$, or $0.01 \cdot 10^1$ and so forth.

The set $\mathbb{F}$ of Octave

Octave employs the set $\mathbb{F}(2, 53, -1021, 1024)$, so:

- base $\beta = 2$;
- $t = 53$ digits for the representation of the mantissa in base 2, corresponding to about 15 digits in base-ten representation;
- $L = -1021$ and $U = 1024$.

Floating-point numbers are distributed symmetrically about zero. They are not evenly spread though, since they become denser as we approach zero and coarser as we move away from zero: see Fig. 3.8, where floating-point numbers are (ideally) represented by vertical dashes.

The smallest positive number in $\mathbb{F}$ for Octave is

$$x_{min} = \beta^{L-1} = 2.225073858507201 \cdot 10^{-308}$$

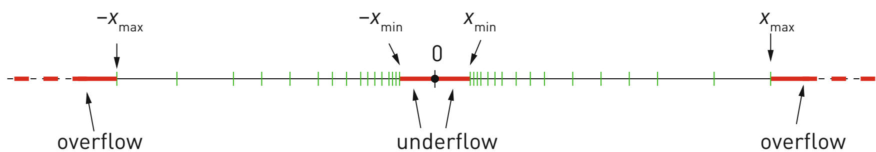


Fig. 3.8 Representation of floating-point numbers on the line

while the largest is

$$x_{max} = \beta^U(1 - \beta^{-t}) = 1.797693134862316 \cdot 10^{+308}.$$

Properties of the set $\mathbb{F}$

The set $\mathbb{F}$ is bounded (hence it does not contain arbitrarily large numbers in absolute value, either positive or negative) and finite, meaning it has a finite number of elements. The immediate consequence is that $\mathbb{F}$ cannot contain all of the reals.

When, during a calculation, we generate a positive number smaller than x_{min}, or negative and larger than $-x_{min}$, we speak about *underflow*, whereas if we generate a positive number larger than x_{max} or negative and smaller than $-x_{max}$ we speak of *overflow*.

3.2.2 Roundoff Errors

Consider a real number x: it may or may not belong to $\mathbb{F}$, hence it may or not be a floating-point number.

If x is a number in $\mathbb{F}$, it means that we can store it in a memory register and the processing unit can perform on it the required operations.

What happens instead if a real number x does not belong to $\mathbb{F}$?

Take for example $x = \pi$ or $x = \frac{1}{10}$: neither belong to $\mathbb{F}(2, 53, -1021, 1024)$ since their exact base-two representation would need more than 53 digits, infinitely many to be precise!

Computer precision

When $x \in \mathbb{R}$ but $x \notin \mathbb{F}$, the computer generates the number $fl(x) \in \mathbb{F}$ (read: *float of* x), which is an *approximation* of x *by rounding*.

The *roundoff error* made to approximate x by its float is

$$\frac{|x - fl(x)|}{|x|} \leq \frac{1}{2}\epsilon_M \tag{3.3}$$

where

$$\epsilon_M = \beta^{1-t} \tag{3.4}$$

is called *machine epsilon* or *computer precision*.

In Octave we have

$$\epsilon_M = 2^{-52} \simeq 2.2204 \cdot 10^{-16},$$

precisely the number we found in problem *Computer precision assessment* regarding the computer precision.

This means that when we employ a computer we will not have the exact value of π but just an approximation (namely, $fl(\pi)$), and with the latter we are making an error of $\frac{\epsilon_M}{2} \simeq 1.1102 \cdot 10^{-16}$ more or less.

Luckily for us, the roundoff error is small and depends exclusively on the base of the representation β and on the parameter t (the number of digits used to store the mantissa of $fl(x)$).

Observe furthermore that the roundoff error $\frac{|x-fl(x)|}{|x|}$ is a *relative error* in x. Moreover, it is certainly more significant than the *absolute error* $|x - fl(x)|$, since the latter does not account for the order of magnitude of x.

The roundoff unit

The number

$$\mathrm{u} = \frac{1}{2}\epsilon_M \tag{3.5}$$

is the maximum relative error the computer can make when representing a real number. For this reason u is called the *roundoff unit*.

In Octave $\mathrm{u} \simeq 1.1102 \cdot 10^{-16}$.

Roundoff errors are not only generated when we input data. They can also arise when we perform operations between floating-point numbers, since nothing guarantees that the outcome of an operation of two floating-point numbers of t digits will still be a floating-point number of t digits.

Example: Roundoff errors in operations

Consider the system $\mathbb{F}(10, 5, -3, 4)$. The computer precision (3.4) and the roundoff unit (3.5) of this system are:

$$\epsilon_M = \beta^{1-t} = 10^{1-5} = 10^{-4}, \qquad \mathrm{u} = \frac{1}{2}\epsilon_M = 0.5 \cdot 10^{-4}.$$

We set out to find the exact sum of $x = 0.12345 \cdot 10^{-3}$ and $y = 0.23821 \cdot 10^{-1}$.

To add numbers in exponential form, the first thing to do is make the exponents equal. More precisely, we transform x so that its exponent becomes the same as that of y, and then we add the decimal parts, like this:

$$\begin{array}{rl} x = & 0.0012345 \cdot 10^{-1} \\ y = & 0.23821 \quad\;\; \cdot 10^{-1} \\ \hline x + y = & 0.2383445 \cdot 10^{-1} \end{array}$$

Clearly $(x + y)$ has 7 decimal digits and so it cannot belong to $\mathbb{F}(10, 5, -3, 4)$, whose elements only have 5 decimal digits. Therefore $(x + y) = 0.2383445 \cdot 10^{-1}$ is approximated by its float $fl(x + y) = 0.23834 \cdot 10^{-1}$ with a roundoff error less than the roundoff unit $\mathrm{u} = 0.5 \cdot 10^{-4}$.

Exact Arithmetic and Floating-Point Arithmetic

Floating-point arithmetic is thus called in opposition to so-called *exact arithmetic*, which is based on the exact execution of elementary operations (hence ignoring roundoff errors) on exact terms (not given by floating-point representations).

Non–associativity in $\mathbb{F}$

In floating-point arithmetic, addition and multiplication are not associative, because it is not always true that $a + (b + c) = (a + b) + c$ or $a \cdot (b \cdot c) = (a \cdot b) \cdot c$.

We have already seen an example where the associativity of addition fails. Here is another one. Consider the numbers `a=1.0e+308`, `b=1.1e+308` and `c=-1.001e+308`. By computing their sum in two different ways we get different results:

```
a + (b + c) = 1.0990e+308,   (a + b) + c = Inf
```

`Inf` (infinity) is the Octave answer when overflow occurs, i.e. when a number is larger than $x_{max} \simeq 1.7976\ldots \cdot 10^{308}$.

Many zeros in $\mathbb{F}$

In floating-point arithmetic the zero is no longer unique, meaning that there are non-zero floating-point numbers that behave like 0 in sums.

Let us clarify this by an example. Let us add the roundoff unit $\mathtt{u} = 1.1102 \cdot 10^{-16}$ to 1.

Compute the sum $1 + \mathtt{u}$ exactly by hand. Let us argue as in example *Roundoff errors in operations*, by thinking in base 10 and supposing mantissas are 15-digit long (we should work in base 2 with 53 digits, but the outcome will nevertheless be significant in base 10 as well). Write 1 in floating-point notation and transform u to have the same exponent as 1:

$$\begin{array}{r@{\,=\,}l}
1 & 0.100000000000000 \qquad\quad \cdot 10^1 \\
\mathtt{u} & 0.00000000000000011102 \cdot 10^1 \\
\hline
1+\mathtt{u} & 0.10000000000000011102 \cdot 10^1
\end{array}$$

Since our floating-point numbers have 15 decimal digits only we deduce $(1 + \mathtt{u})$ cannot be in $\mathbb{F}$, and so $(1 + \mathtt{u})$ gets approximated by its float (with only 15 digits starting from the first non-zero one) $fl(1 + \mathtt{u}) = 0.100000000000000 \cdot 10^1 = 1$.

Hence $1 + \mathtt{u} = 1$, but $\mathtt{u} \neq 0$.

We have thus found a number $\mathtt{u} \neq 0$ that added to 1 still gives 1, which means that it behaves like zero *even though it is not zero.* And just like u, there exist a myriad of other numbers with this property.

You might ask: why the need to store the number u when it is so small that it is, to all effects, irrelevant when we add it to 1? And furthermore, why does the computer store numbers that are even smaller than u?

Everything Is Relative

The answer is simple. Although u is in practice zero when we add it to 1 (or to any number larger than 1), this number is more important than any other number smaller than itself, and it is not irrelevant at all if we add it to numbers less than 1. For example the Octave instructions:

```
format long e
u=1.1102e-16;
x=3.e-4;
x+u
```

will give

```
ans = 3.00000000000111e-04
```

which is different from `x=3.00000000000000e-04`.

Propagation of Roundoff Errors

Although roundoff errors are very small (they have order 10^{-16}), it might happen that they are repeated several times in long and complex algorithms, and amplified to the point of producing an effect that cannot be neglected.

A Recursive Formula for Computing π

Consider the sequence[7] of real numbers a_n, with $n \geq 1$, defined recursively as follows:

$$\begin{aligned} a_1 &= 2 \\ a_2 &= 2^{\frac{3}{2}}\sqrt{1-\sqrt{1-4^{-1}a_1^2}} \\ a_3 &= 2^{\frac{5}{2}}\sqrt{1-\sqrt{1-4^{-2}a_2^2}} \\ &\dots \\ a_n &= 2^{n-\frac{1}{2}}\sqrt{1-\sqrt{1-4^{1-n}a_{n-1}^2}} \end{aligned}$$

It can be proved[8] that the values a_n tend to π as n goes to infinity, i.e.

$$\lim_{n\to\infty} a_n = \pi.$$

We want to check whether this noteworthy result holds when we use the computer.

Beside computing some elements of the sequence a_n, we also compute the elements of this other sequence:

$$e_n = \frac{|a_n - \pi|}{\pi}, \qquad \text{for } n \geq 1.$$

---

[7]A *sequence* is a real-valued function with domain in the natural numbers. The writing $y = a_n$ is used in place of the typical notation $y = a(n)$ used for functions of a real variable.

[8]This was done by the French mathematician François Viète (1540–1603).

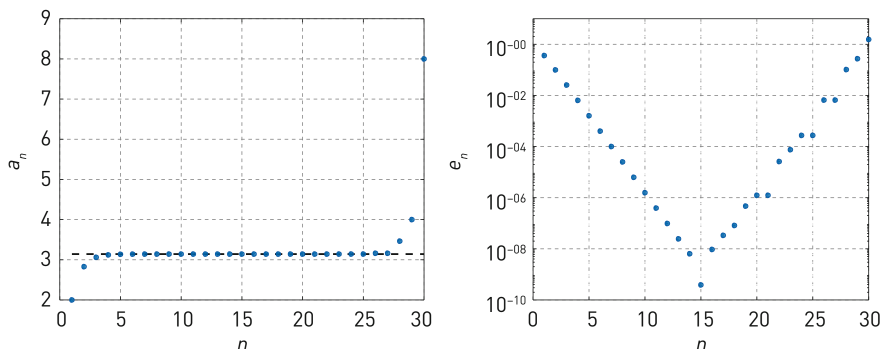


Fig. 3.9 The first 30 values of the sequences a_n (left) and e_n (right)

Taking for example $n = 5$, the value $e_5 = \dfrac{|a_5 - \pi|}{\pi}$ gives a (relative) measure of how well $a_5 = 3.13654849054594$ approximates π. In other words if we were satisfied to approximate π by a_5 we would make the relative error

$$e_5 = \frac{|3.13654849054594 - \pi|}{\pi} \simeq 1.6056 \cdot 10^{-3}.$$

Since we know that $\lim_{n\to\infty} a_n = \pi$, it follows that $\lim_{n\to\infty} e_n = 0$, i.e. we expect the sequence e_n to become smaller and smaller as n increases.

We have prepared the script `pigreek` (see script 3.1 on p. 95) to compute the first 30 numbers of the sequences a_n and e_n.

Typing

```
pigreek
```

in the command window of Octave, we obtain Fig. 3.9.

While to represent a_n (with $n = 1, \ldots, 30$) we used the `plot` command, to represent e_n we used the command `semilogy`, which generates a graph with linear scale along the x-axis and logarithmic scale along the y-axis (see Chap. 6, Sect. 6.4.5 for the logarithmic scale).

`semilogy`

It seems that the sequence a_n is approaching π as long as $n \leq 25$, and then departs from π when $n > 25$.

The right picture in Fig. 3.9 is even more explanatory: the errors e_n decrease as $n \leq 15$ to around 10^{-10}, and then start to grow for $n > 15$ to reach the value 1.5465 when $n = 30$. These values should tend to zero as n goes to $+\infty$, and instead they are moving away from zero.

Despite the fact that it can be proved that the sequence a_n converges to π as n tends to $+\infty$, what we obtain numerically is a completely different story!

The reason for this behaviour should be ascribed to the *propagation of roundoff errors*, and in particular to the so-called *cancellation errors* made by machine arithmetic when adding quantities with opposite sign and very close absolute value.

3.3 What We Have Learnt

We have learnt to work with Octave: performing operations with numbers, plotting functions, manipulating matrices and vectors, writing short programs.

We have also understood how computers represent numbers, how *roundoff errors* are generated, and that *machine arithmetic* is at times rather different from the exact arithmetic we have been studying for a long time.

Readers who have stuck by until this moment have invested on a skill that will pay off generously in the future.

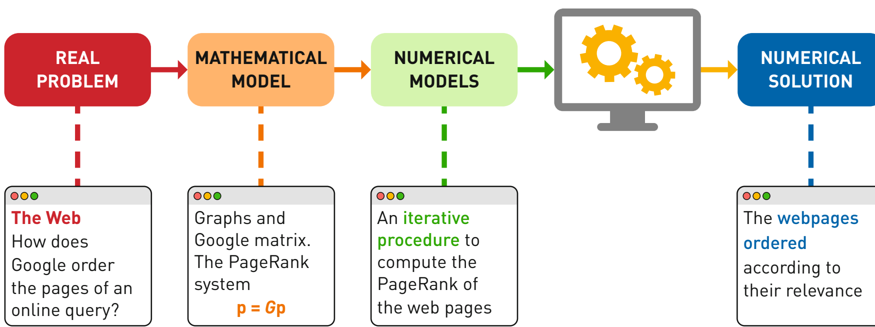
3.4 Script of This Chapter

```
% script pigreek
a=zeros(30,1); e=zeros(30,1); % initialises the vectors a and e
% they will contain the first 30 elements
% of the sequences a_n and e_n
a(1)=2; e(1)=abs(pi-a(1))/pi; % first element of the sequences
for n=2:30    % loop for generic n
  a(n)=2^(n-0.5)*sqrt(1-sqrt(1-4^(1-n)*a(n-1)^2));
  e(n)=abs(pi-a(n))/pi;
end
% plot the sequence  a_n
figure(1); clf
N=1:30; plot(N,a,'bo','Linewidth',2); hold on
plot([1,30],[pi,pi],'k--') % plots the line y=pi
xlabel('n'); ylabel('a_n'); grid on
% plot the sequence e_n
figure(2); clf
N=1:30;
semilogy(N,e,'bo','Linewidth',2)  % log scale in y,
                                  % linear scale in x
xlabel('n'); ylabel('e_n'); grid on
```

Script 3.1: `pigreek.m`: Script for computing π using the sequence a_n of p. 93.

A Virtual Surfer for the Web

4



INGREDIENTS	matrices, matrix-vector product, definition of probability.
WHAT WE LEARN	matrix of a graph, iterative method, stopping test.
PREREQUISITES	Chapter 1, Sect. 2.2 of Chap. 2. Ability to use Octave for simple one-line commands (no deep knowledge necessary; download of the package of Octave functions and scripts required). Section 3.1, Chap. 3 needed to understand and learn the content of Octave functions and scripts.

4.1 Surfing with Google

It is the 22nd of May and we decide to look up born on 22 May on Google. In the blink of an eye, 0.35 seconds to be precise, we get back around 624,000 hits. We click on the first link Google proposes (22 May – Wikipedia) and discover that on this very day in 1859 Sir Arthur Conan Doyle was born in Edinburgh, Scotland (see Fig. 4.1).

Although it would not have pleased Sir Arthur, who would rather have been remembered more for his science fiction than for the crime novels, our thought

© Springer Nature Switzerland AG 2020

A. Quarteroni, P. Gervasio, *A Primer on Mathematical Modelling*, UNITEXT 121,
https://doi.org/10.1007/978-3-030-44541-6_4

Fig. 4.1 Sir Arthur Conan Doyle. (Photo by Walter Benington from Wikimedia, 1914, public domain)

rushes to Sherlock Holmes. So we type Sherlock Holmes in the search bar and Google finds about 2,170,000 results in 0.84 seconds. Among the suggested links we opt for the one mentioning the actor Robert Downey Jr., and from there we go to Avengers: Infinity War and then ...

...and then we are forced to return to this book, but in the meantime we have made a fantastic mind journey and Google has trailed (anticipated?) us almost as fast as the thoughts taking shape in our minds.

More generally, when we type one or more keywords in the Web[1] using a search engine like Google, in a very short time, often less than a second, we get a list of webpage addresses called URLs (*Uniform Resource Locator*) related to the keywords we have provided.

If we think of the immense quantity of information present on the Web, that is already noteworthy.

Even more shocking is discovering that the addresses are listed in an order that reflects the relevance of the pages and their pertinence to our query.

Truth be told, we are unconsciously aware of this ordering, since we often simply pick the first link that shows up rather than check the thousand other addresses found by the search engine.

How does Google answer so quickly to our requests? And most of all, how does it find and organise the pages relevant to our search terms? By which criterion are the pages listed?

4.2 Understanding the Problem

The Web may be compared to a massive book, one that is very chaotic and under constant update.

[1] Web stands for *World Wide Web*, a service available on the internet that allows to access a gigantic quantity of written and multi-media information.

A search engine has the following purposes:

- build a very detailed analytical index[2] of this book;
- keep it updated;
- be able to browse through it very quickly;
- produce a list of the pages related to our query, ordered according to a suitable criterion.

Searching the Web is the same as consulting the analytical index of this book.

Scanning the Web

To build the index and keep it up to date, Google first of all constantly scans the Web with a special software called Googlebot. It finds new and updated pages (which differ from those of earlier scans), recognises the links inside of them and adds all the information to the list of pages to be examined more carefully with regard to the keywords. It is a long and repetitive job, made by an immense number of computers.

Indexing Content

The second phase is the indexing of the various contents. Googlebot processes each page in the list built during the scanning phase, with the aim of compiling and updating the index of all words that were found, and memorise their position within each page.

Ordering Pages

At last, when we submit a query, Google's computers look inside this enormous index for the pages related to the word (o words) we have typed, and they produce the list of the pages considered most significant, ordered by relevance.

The *pertinence* of a page (the relationship between the page and the word(s) we are looking for) and its *relevance* (its importance or reliability, relative to all other pages on the Web) are established on the basis of over 200 factors. One of the factors involved in assessing the relevance of a page is called *PageRank*.

Here we are interested in this last phase of the work, namely when Google orders the pertinent pages.

First of all, let us recall how one defines an *ordering in a set*.

To list the objects of an arbitrary set by importance (here, webpages), we must attach to each object a real number that will measure its importance, according to some criteria given in advance. Only then we will be able to create a ranking.

Google uses an algorithm to organise webpages that is mostly secret. Only certain bits are made public, and usually this happens when they have become obsolete. Clearly we are not here to blow the whistle on Google's secrets (which we do not know), but we shall content ourselves with describing the part of the algorithm that is public.

[2] An analytical index is a reference index (typically organised alphabetically) of the keywords appearing in a book. In correspondence to each word it gives the page numbers where that word appears in the text. The analytical index at the end of this book is an example.

4.2.1 The PageRank

The PageRank ranking (or just PageRank) of a webpage is a number that encodes the relevance of the webpage measured against all other webpages. The higher its PageRank, the more relevant the webpage is.

Each webpage examined by Google[3] has its own PageRank. It is calculated using a sophisticated algorithm (called *PageRank algorithm*) independently of the specific words used in the query. It is also continuously updated in terms of the evolution of the Web.

In order to generate the ranking of all webpages, Google's PageRank algorithm simulates the behaviour of a virtual *surfer* that browses randomly and relentlessly the Web, making searches and following a trail of links from one page to the next.

PageRank is a probability

The PageRank of a given webpage represents the *probability*[4] that an arbitrary user lands on that webpage by browsing the Web randomly.

Let us remark that a PageRank ranking is an index that only takes into consideration the structure of the Web (that is, how webpages are connected by links). Its scope is to help Google address our queries in the fastest-possible and most appropriate way.

The PageRank does not assess the contents of a page, it does not look at the truthfulness of the data, nor does it judge the scientific, political or cultural reputation of the author of the information, or the meaning or ethical standing of what is written.

A Computer for Searching, Many Computers for Finding

It is clearly hopeless to even think of computing the PageRank of 20 billion pages by hand. What is worse, even a personal computer like the one we use every day is guaranteed to incur serious problems in attempting such a feat.

What is needed is several computers with a large working memory (the RAM, *Random Access Memory*), a large mass memory (such as storage discs), and equipped with many processors, each performing billions of elementary operations per second and working in parallel.

Note further that Google repeats these computations very frequently, so to always have super-updated PageRank numbers, especially concerning political events, sports or the news.

[3] As of August 2020 the total number of webpages ranked by Google was around 60 billion (source http://www.worldwidewebsize.com).

[4] The probability p that an event occurs is the ratio of the number of occurrences of the event to the total number of events. Therefore a probability is always a real number $0 \leq p \leq 1$. Saying $p = 0$ means the event is impossible, $p = 1$ means it will occur with certainty, while $p = 0.5$ means it has a 50% chance of happening.

In the next pages we will see how to construct a mathematical model and then a numerical model to determine the PageRank of simple networks. Once we have that, we will be able to list webpages by PageRank, starting with the ones with highest ranking.

4.3 Simplify in Order to Model

Consider a tiny network of 4 webpages connected by links, as shown in Fig. 4.2.

4.3.1 The Directed Graph as a Model of the Web

The network is modelled as a *directed graph*, which is a collection of elements (called *nodes*) and connections (called *edges*) oriented from one node to another. Each node represents a webpage and each edge is a *link* between two webpages.

The directed graph associated with the network of Fig. 4.2 is made of 4 nodes, see Fig. 4.3.

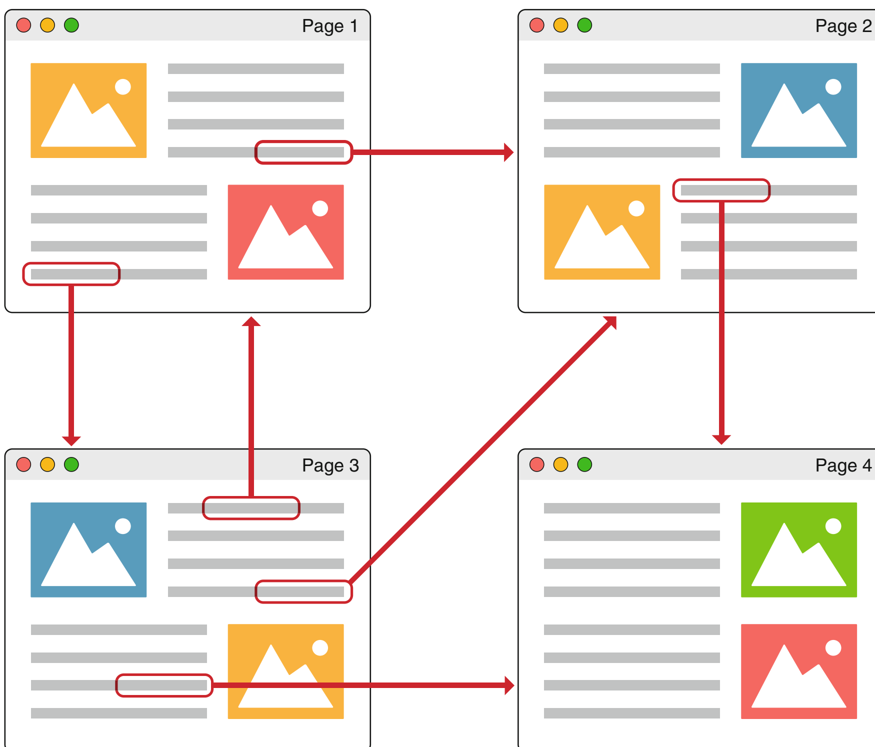


Fig. 4.2 A network of 4 webpages and the links between them

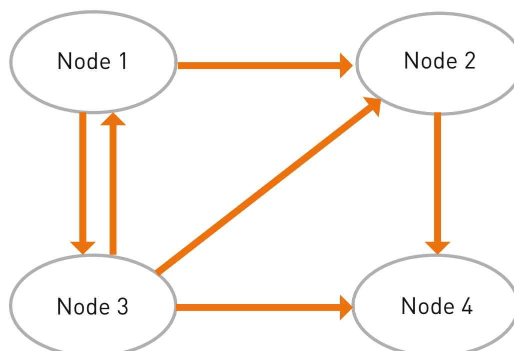


Fig. 4.3 The directed graph modelling the network of Fig. 4.2

4.3.2 The Adjacency Matrix of a Graph

The information of the graph, i.e. the number N of nodes and the links, can be stored in a matrix A called the *adjacency matrix of the graph*.

Entries of the adjacency matrix

The adjacency matrix A of a graph has N rows and N columns, as many as the number of nodes. Its elements a_{ij} (with $i = 1, \ldots, N$ and $j = 1, \ldots, N$) are defined by:

$$a_{ij} = \begin{cases} 1 \text{ if there is a link from node } j \text{ to node } i, \\ 0 \text{ otherwise.} \end{cases} \tag{4.1}$$

Exercises

Solve Exercise 4.1, p. 216 and Exercise 4.2, p. 217.

For example, the matrix A associated with the graph of Fig. 4.3 is the following square matrix of order $N = 4$:

$$A = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{bmatrix}.$$

The first column stores the information concerning the links originating from the first node and pointing to other nodes. By inspecting A we deduce that two links emerge from node 1: one towards node 2 (so $a_{21} = 1$) and one towards node 3 ($a_{31} = 1$).

In the second column of A there is only one non-zero element, the fourth one. In fact there is just one link departing from the second node, which reaches node 4 (whence $a_{42} = 1$).

The first, second, and last elements of column three are non-zero, so from node 3 there are three outgoing links, pointing to each of the other pages of the network.

The fourth column has all zeroes, reflecting the fact that node 4 has no outgoing link. We will say that this is a *dangling* node.

Although the adjacency matrix A retains the essential information of the network, it is not suitable to simulate a virtual surfer hopping across the Web. For that we need to introduce other matrices.

4.4 From the Adjacency Matrix *A* to the Google Matrix *G*

Let us indicate by L_j the number of outgoing links from node j, for every $j = 1, \dots, N$. We define a new adjacency matrix H of order N, whose elements h_{ij} ($i = 1, \dots, N$, $j = 1, \dots, N$) are:

$$h_{ij} = \begin{cases} \dfrac{1}{L_j} & \text{if } L_j \neq 0 \text{ and there exists a link from node } j \text{ to node } i, \\ 0 & \text{otherwise.} \end{cases} \tag{4.2}$$

Referring to the graph of Fig. 4.3: $L_1 = 2$, $L_2 = 1$, $L_3 = 3$ and $L_4 = 0$. Using formula (4.2) we obtain the matrix

$$H = \begin{bmatrix} 0 & 0 & \frac{1}{3} & 0 \\ \frac{1}{2} & 0 & \frac{1}{3} & 0 \\ \frac{1}{2} & 0 & 0 & 0 \\ 0 & 1 & \frac{1}{3} & 0 \end{bmatrix}.$$

Note that the elements of H are real numbers between 0 and 1 (possibly 0 or 1 too) and the sum of the elements in a column always equals 1, except for column 4 (associated with a *dangling* node) where the sum is zero.

The Probability of Clicking on a Link

A matrix of probabilities

The element h_{ij} of matrix H represents the probability that a virtual surfer that is on page j of the network decides to click on the link that takes him from page j to page i.

Suppose, for example, the surfer is visiting page 3 of the network of Fig. 4.2: the probability that he clicks on any of the three links on page 3 is $1/3$, since the elements h_{13}, h_{23} and h_{43} are all equal to $1/3$.

Assuming the surfer chooses page 1, he would then have probability $1/2$ to jump to page 2 or 3 next, and so on.

It is useful to note that the matrices A and H say nothing about the content, but speak only about the connections between the various pages. They cannot interpret the thoughts or preferences of the surfer.

As in a Black Hole

The matrix H has a problem, though: if the surfer could move uniquely along the trail of links described by the graph through H, once he arrived on page 4 of the network in Fig. 4.2 he would end his journey, because page 4 has no outgoing link.

A webpage with no outgoing links (a *dangling* node in the graph) behaves like a black hole! How does one get out of it?

A Web user that reaches a page without links and is not tired of surfing, typically types in the *browser*[5] bar a known URL address, or starts a new Google search with other keywords. To simulate this behaviour we should substitute the matrix H by a more complete matrix.

"If You Feel You Are in a Black Hole, Don't Give Up. There's a Way Out" (S. Hawking)

Let us suppose the surfer has reached page 4 of the network of Fig. 4.2 and decides to type a new URL, or make a new query.

To simulate the action of leaving page 4 for another page we must replace the zeroes in column four of H with certain numbers indicating the probability of moving from page 4 to the other pages. Since a priori there are no preferred pages, we will assign to each page the same probability $1/N$.

Hence, we replace H by a new matrix S (of the same order N) such that:

- *if* j is a *dangling* node, all elements of column j are set equal to $1/N$;
- *otherwise* we set $s_{ij} = h_{ij}$, i.e. we keep the information of H.

More precisely, and without referring to H, the elements s_{ij} ($i = 1, \ldots, N$ and $j = 1, \ldots, N$) of S are as follows:

$$s_{ij} = \begin{cases} \dfrac{1}{N} & \text{if } L_j = 0 \text{ (} j \text{ is a \textit{dangling} node),} \\ \dfrac{1}{L_j} & \text{if } L_j \neq 0 \text{ and there is a link from node } j \text{ to node } i, \\ 0 & \text{if } L_j \neq 0 \text{ and there is no link from node } j \text{ to node } i. \end{cases} \tag{4.3}$$

[5] A browser is a program allowing to navigate the Web. Examples include Chrome, Firefox, Safari or Internet Explorer.

The matrix S associated with the network of Fig. 4.2 is then

$$S = \begin{bmatrix} 0 & 0 & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{2} & 0 & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{2} & 0 & 0 & \frac{1}{4} \\ 0 & 1 & \frac{1}{3} & \frac{1}{4} \end{bmatrix}.$$

Now the sum of the elements on any column (including column 4, associated with the *dangling* node) equals 1.

The Possibility of Changing Path

When a user surfs the Web he actually does not merely follow links. All of a sudden he might decide to type the URL of a different page in the browser, or initiate a new search in Google. This is exactly what happened when we decided not to choose any link appearing on page 22 May - Wikipedia, and instead we started a new query using the words Sherlock Holmes.

The Google matrix

To simulate this change of path, as well, we introduce a new parameter α (a real number with $0 \leq \alpha < 1$), and define a matrix G called the *Google matrix*. Its elements (for $i = 1, \ldots, N$ and $j = 1, \ldots, N$) are:

$$g_{ij} = \alpha s_{ij} + \frac{1-\alpha}{N}. \tag{4.4}$$

The element g_{ij} expresses the probability a user, currently visiting page j, moves from page j to page i.

Notice that the g_{ij} are all real numbers between 0 and 1, and the sum of the elements on a column of G is always 1:[6]

$$\begin{array}{ll} 0 \leq g_{ij} \leq 1 & \text{for every } i = 1, \ldots, N, \ j = 1, \ldots, N, \\ g_{1j} + g_{2j} + \ldots + g_{Nj} = 1 & \text{for every } j = 1, \ldots, N. \end{array} \tag{4.5}$$

Choosing $\alpha = 0$ gives $g_{ij} = \dfrac{1}{N}$ for every pair i and j of nodes. In this case, therefore, the matrix G models a surfer hopping from page to page always with the same probability, irrespective of the presence of links.

---

[6] A matrix with this property is called a *column-stochastic matrix*.

When α is different from zero, the larger α is, the higher the probability a user will follow a path of links, as opposed to starting a new search from scratch. A common value is $\alpha = 0.85$.

4.5 The Mathematical Model

Let us denote by p_i the PageRank of page i. Since PageRanks represent probabilities, we have

$$0 \leq p_i \leq 1, \qquad \text{for } i = 1, \ldots, N. \tag{4.6}$$

We further impose

$$p_1 + p_2 + \ldots + p_N = 1. \tag{4.7}$$

How can we compute the PageRanks $p_1, p_2, \ldots, p_N$ starting from the information stored in the matrix G?

Let us focus on the PageRank of the second webpage of the network of Fig. 4.2. The virtual surfer might reach page 2 because of a link present on another page, or by starting a new search starting from page 2.

It's a Matter of Probability

Suppose the surfer has reached page 2 by clicking on a link appearing on page 1. We know the element g_{21} of G expresses the probability that the surfer, assumed to be on page 1 of the network, decides to jump to page 2.

However, the fact that he currently is on page 1 is not a certain event, for it itself has a probability of occurring, which is precisely the PageRank p_1.

Conditional probability

The probability the surfer lands on page 2 starting from page 1 is a *conditional probability*, meaning it depends on him already being on page 1. This conditional probability equals the product $g_{21} p_1$.

Similarly, $g_{22} p_2$ expresses the probability to stay on page 2; $g_{23} p_3$ expresses the probability to reach page 2, under the condition that he starts from page 3; $g_{24} p_4$ is the probability to reach page 2 supposing he was on page 4.

Therefore the *total probability of visiting page 2* is given by the sum of all conditional probabilities listed above:

$$p_2 = g_{21} p_1 + g_{22} p_2 + g_{23} p_3 + g_{24} p_4.$$

This formula follows from an important theorem called the *law of total probability*. The fundamental hypothesis used to apply this theorem is to assume the surfer cannot visit at the same time two different pages. In other words, the fact of being on one webpage is *incompatible* with being simultaneously on another page.

Now we can repeat the same argument for the PageRanks of the other pages of the small network of Fig. 4.2 and obtain the system of equations

$$\begin{cases} p_1 = g_{11}p_1 + g_{12}p_2 + g_{13}p_3 + g_{14}p_4 \\ p_2 = g_{21}p_1 + g_{22}p_2 + g_{23}p_3 + g_{24}p_4 \\ p_3 = g_{31}p_1 + g_{32}p_2 + g_{33}p_3 + g_{34}p_4 \\ p_4 = g_{41}p_1 + g_{42}p_2 + g_{43}p_3 + g_{44}p_4. \end{cases}$$

Each PageRank depends on the PageRanks of all the pages of the network.

More generally, by repeating the argument for a network of dimension N, we arrive at the following system of linear equations, whose unknowns $p_1, p_2, \ldots, p_N$ are the PageRanks of the N pages of the network:

$$\begin{cases} p_1 = g_{11}p_1 + g_{12}p_2 + \ldots + g_{1N}p_N \\ p_2 = g_{21}p_1 + g_{22}p_2 + \ldots + g_{2N}p_N \\ \ldots \\ p_N = g_{N1}p_1 + g_{N2}p_2 + \ldots + g_{NN}p_N. \end{cases} \tag{4.8}$$

Let us now use matrices and vectors to write in a more compact form the linear system (4.8) just obtained.

Matrices and Vectors to Help Us Model

Let us store the PageRanks $p_1, \ldots, p_N$ in a column vector $\mathbf{p}$ (called the *PageRank vector*)

$$\mathbf{p} = \begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_N \end{bmatrix}$$

and remember the matrix G is

$$G = \begin{bmatrix} g_{11} & g_{12} & \cdots & g_{1N} \\ g_{21} & g_{22} & \cdots & g_{2N} \\ \vdots & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ g_{N1} & g_{N2} & \cdots & g_{NN} \end{bmatrix}.$$

Recalling the matrix-vector product seen on page 35, we observe that the right-hand-side terms of system (4.8) are exactly the components of the product between the matrix G and the vector $\mathbf{p}$. Consequently, we may rewrite the N equations of system (4.8) in the more compact form $\mathbf{p} = G\mathbf{p}$, which encompasses them all.

We've Made It

The mathematical model for PageRanks

The PageRanks of an N-page network are the components of the vector $\mathbf{p}$ solution to system

$$\mathbf{p} = G\mathbf{p}.$$

This system, here in matrix form, is equivalent to system (4.8), and represents our *mathematical model* for computing the PageRanks.

Since G satisfies properties (4.5), the equation[7] $\mathbf{p} = G\mathbf{p}$ (or, equivalently, system (4.8)) admits a unique solution.[8] That is, there exists a unique PageRank vector $\mathbf{p}$, whose components satisfy conditions (4.6) and (4.7), that solves $\mathbf{p} = G\mathbf{p}$.

Referring to problem *Google* we thus have:

- the *datum of the problem* is the graph (or its adjacency matrix) associated with the Web;
- the *mathematical model* is the system $\mathbf{p} = G\mathbf{p}$ in matrix form;
- the *solution* is the PageRank vector $\mathbf{p}$ of the pages in the network.

4.6 The Numerical Model: How Do We Solve the System p = Gp?

System $\mathbf{p} = G\mathbf{p}$ is not a standard system of linear equations, for the unknown vector $\mathbf{p}$ appears on both sides of the equation, so $\mathbf{p}$ is known only implicitly in terms of itself. Let us present one possible strategy to compute $\mathbf{p}$ numerically. We begin by defining the N-dimensional column vector

$$\mathbf{p}^{(0)} = \begin{bmatrix} \frac{1}{N} \\ \frac{1}{N} \\ \vdots \\ \frac{1}{N} \end{bmatrix}$$

and compute the product $\mathbf{p}^{(1)}$ between the matrix G and the vector $\mathbf{p}^{(0)}$:

$$\mathbf{p}^{(1)} = G\mathbf{p}^{(0)}.$$

[7] Note that $\mathbf{p} = G\mathbf{p}$ can be called either "equation" (it is a matrix equation because it involves matrices and vectors) or "system" (meaning the system of equations (4.8)).

[8] The proof of this important result requires one to know about the *eigenvalues* and *eigenvectors* of G, which are mathematical entities studied at university.

Once we have $\mathbf{p}^{(1)}$, we compute the product $\mathbf{p}^{(2)}$ of G and $\mathbf{p}^{(1)}$:

$$\mathbf{p}^{(2)} = G\mathbf{p}^{(1)},$$

and then, given $\mathbf{p}^{(2)}$, we compute

$$\mathbf{p}^{(3)} = G\mathbf{p}^{(2)}$$

and so on.

The iterative process

These operations can be summarised by the *iterative process* (regarding *for loops* see page 39):

$$\begin{aligned}&\text{given } \mathbf{p}^{(0)},\\ &\textbf{for } k = 1, 2, 3, \ldots \textbf{ do}\\ &\quad \text{compute } \mathbf{p}^{(k)} = G\mathbf{p}^{(k-1)}\\ &\textbf{end}\end{aligned} \tag{4.9}$$

For starters, the initial choice of vector $\mathbf{p}^{(0)}$ with all components equal to $1/N$ is saying that we assume the PageRanks are all the same, and therefore that each webpage in an N-page network can be visited with the same probability $1/N$.

At each iteration k of process (4.9), the new vector $\mathbf{p}^{(k)}$ is a better approximation than $\mathbf{p}^{(k-1)}$ of the solution vector $\mathbf{p}$.

This is a bit like when we prepare a milkshake for the first time, and we do not know for how long we should keep mixing the ingredients (see Fig. 4.4). The vector $\mathbf{p}^{(0)}$ is the mixture of fruit, milk and sugar. The vector $\mathbf{p}^{(1)}$ is the result of the first attempt: if we are satisfied with it we stop, otherwise we take what we have ($\mathbf{p}^{(1)}$) and mix another while to get $\mathbf{p}^{(2)}$; then we mix the compound $\mathbf{p}^{(2)}$ and get $\mathbf{p}^{(3)}$ and so forth, until we believe we have got what we wanted.

Fig. 4.4 Milkshakes. (Photo by Silvia Schütz from Pixabay, 2017; id 2253423)

Is the Milkshake Ready? The Stopping Test

When we prepare a milkshake we might are able to decide when to stop simply by looking at the result (exercising the *sense* of *sight*) or tasting it (exercising the *sense* of *taste*).

Understanding when to stop the iterations in (4.9) when solving system $\mathbf{p} = G\mathbf{p}$ is a more delicate matter. Here, too, we must invoke a suitable *"mathematical sense"*, to devise a *stopping test*. The latter allows us to understand whether we can stop process (4.9) or, rather, we should increase the index k and compute the next vector $\mathbf{p}^{(k)}$.

How many vectors $\mathbf{p}^{(k)}$ must we compute? Which tests can we employ to stop the iterations?

Because we do not know the exact PageRanks $p_1, p_2, \ldots, p_N$, it is clear that we cannot say *"we stop when the values $p_1^{(k)}, p_2^{(k)}, \ldots, p_N^{(k)}$ are sufficiently close to the corresponding (unknown) values $p_1, p_2, \ldots, p_N$"*. We cannot, in other words, compute the error between the solution $\mathbf{p}^{(k)}$ found at iteration k and the exact solution $\mathbf{p}$.

However, we can define an *error estimator* (here's the "mathematical sense"), denoted $E^{(k)}$, as follows. For every step of the loop, i.e. for every value k starting from 1, and for every component of the PageRank vector we compute the absolute value of the difference between the values at iterations k and $k-1$, divided by the value at iteration k:

$$
\begin{aligned}
e_1^{(k)} &= \frac{|p_1^{(k)} - p_1^{(k-1)}|}{p_1^{(k)}},\\
e_2^{(k)} &= \frac{|p_2^{(k)} - p_2^{(k-1)}|}{p_2^{(k)}},\\
&\ldots,\\
e_N^{(k)} &= \frac{|p_N^{(k)} - p_N^{(k-1)}|}{p_N^{(k)}}.
\end{aligned}
$$

Then we compute the maximum among all these

$$
E^{(k)} = \max\{e_1^{(k)}, e_2^{(k)}, \ldots, e_N^{(k)}\}.
$$

At this point there are two possibilities.

- *If* we believe $E^{(k)}$ is small enough, i.e. less than a certain given *tolerance* ϵ[9] (which means the values $p_i^{(k)}$ are very close to the values $p_i^{(k-1)}$, and another iteration would be almost useless), then we stop and *accept the vector* $\mathbf{p}^{(k)}$ *as the approximation of the unknown PageRank vector* $\mathbf{p}$. If so, $\mathbf{p}^{(k)}$ will represent the *numerical solution* to our problem.
- *Otherwise* we increase k and perform another iteration.

[9] In mathematics ϵ usually denotes a positive real number very close to zero.

The choice of tolerance ϵ depends on the accuracy we require in approximating **p**: the smaller ϵ is, the more accurate the computational solution will be.

In principle, choosing $\epsilon = 10^{-3}$ permits us to compute an approximation of the PageRanks p_i that is exact up to the third decimal digit. Similarly, $\epsilon = 10^{-6}$ approximates exactly till the sixth digit, etc.

Apart from checking that $E^{(k)} < \epsilon$, in practice one also checks that k does not exceed a maximum value k_{max} of iterations allowed. This is to avoid iterating too many times due to the propagation of roundoff errors or to the large dimension N of the network under consideration (we do not wish to keep mixing forever a compound that might never become perfectly smooth: imagine if we had used fruit with a hard shell).

4.7 The Algorithm for Computing PageRanks

The following algorithm summarises the operations we have described (note that the *for loop* on the index k and the stopping test explained in the previous section have been substituted by a *while loop*, see the next subsection).

Algorithm 4 *Iterative method for computing PageRanks*

Data: Google matrix G of the network, tolerance ϵ, maximum number k_{max} of iterations
Result: PageRank vector **p**, number k of iterations made
compute the order N of matrix G;
set $E^{(0)} = \epsilon + 1$;
define $\mathbf{p}^{(0)} = [1/N, \ldots, 1/N]$, an N-dimensional column vector;
set $k = 0$;
while $k < k_{max}$ ***and*** $E^{(k)} > \epsilon$ **do**
  increase the index k by one, i.e. set $k = k + 1$;
  compute $\mathbf{p}^{(k)} = G\mathbf{p}^{(k-1)}$;
  compute $E^{(k)} = \max\{|p_1^{(k)} - p_1^{(k-1)}|/p_1^{(k)}, \ldots, |p_N^{(k)} - p_N^{(k-1)}|/p_N^{(k)}\}$;
end
set $\mathbf{p} = \mathbf{p}^{(k)}$;

Once the vector **p** has been calculated we can list pages in decreasing order according to the values p_i.

4.7.1 While Loops in Algorithms

The instructions

while $k < k_{max}$ ***and*** $E^{(k)} > \epsilon$ **do**
  increase the index k by one, i.e. set $k = k + 1$;
  compute $\mathbf{p}^{(k)} = G\mathbf{p}^{(k-1)}$;
  compute $E^{(k)} = \max\{|p_1^{(k)} - p_1^{(k-1)}|/p_1^{(k)}, \ldots, |p_N^{(k)} - p_N^{(k-1)}|/p_N^{(k)}\}$;
end

have the following meaning:

if $k < k_{max}$ and $E^{(k)} > \epsilon$, *do* the following operations:

- increase the variable k by one;
- compute the new PageRank vector $\mathbf{p}^{(k)}$;
- compute the new error $E^{(k)}$;

go back to "if".[10]

The instructions between the words **do** and **end** constitute the *body of the while loop*. They are repeated time and again, so long as the conditions "$k < k_{max}$" and "$E^{(k)} > \epsilon$" are both met.

It is clear that to avoid executing the instructions in the body of the loop over and over without stopping, we should modify the values of k and $E^{(k)}$ before returning to the instruction *"while"* (that is, before the end of the loop). Therefore the instructions modifying the values of k and $E^{(k)}$ must belong in the body of the loop, which is what we did in Algorithm 4.

Exercise: A 4-page network

Compute the PageRanks of the pages of the network in Fig. 4.2. Which page has the highest PageRank?

Solution.

- The *datum of the problem* is Fig. 4.3;
- the *mathematical model* is the matrix equation $\mathbf{p} = G\mathbf{p}$;
- the *numerical model* is the iterative process (4.9), formalised in Algorithm 4;
- the *solution* is given by the PageRanks $p_1, \ldots, p_4$ of the four pages of the network.

We have to follow these steps:

1. build the adjacency matrix A of the graph;
2. given $\alpha = 0.85$, build the Google matrix G by computing its elements g_{ij} according to definition (4.4) on page 105;
3. fix ϵ and k_{max} and apply Algorithm 4 to compute the vector $\mathbf{p}$;
4. order the pages according to their PageRank.

Let us determine the elements of A by inspecting the graph in Fig. 4.3, as we have seen in Sect. 4.3. The Octave command that assigns to the variable `A` the matrix A is

```
A=[0 0 1 0; 1 0 1 0; 1 0 0 0; 0 1 1 0];
```

[10] The initial "if" and the final "go back" instructions are logically equivalent to the word "while". For this reason the loop in question is called a *while loop*.

To construct G we have predefined the function `matrix_G` written in Octave language (see Function 4.1, page 116). After we choose $\alpha = 0.85$, to build `G` it suffices to type the following instruction

```
G=matrix_G(A,0.85)
```

(`A` and `0.85` are the input of the function, `G` is the output). We obtain

```
G =
    0.037500    0.037500    0.320833    0.250000
    0.462500    0.037500    0.320833    0.250000
    0.462500    0.037500    0.037500    0.250000
    0.037500    0.887500    0.320833    0.250000
```

and we can check that all elements of G lie between 0 and 1 and that the sum of the elements on any column equals 1.

To compute the PageRanks we call the `compute_pagerank` function that implements Algorithm 4 (see Function 4.2 on page 117). After we have fixed $\epsilon = 10^{-3}$ (in Octave this is written `1e-3`) and $k_{max} = 100$, by typing the instruction

```
[p,k]=compute_pagerank(G,1e-3,100)
```

the screen shows:

```
p =
    0.17406
    0.24799
    0.19324
    0.38472

k =   6
```

The algorithm has stopped after 6 iterations and has produced the PageRanks of the four pages.

At this point we can order the pages by PageRank and establish their place in the ranking, as shown in Table 4.1.

In conclusion the page with highest PageRank is page 4, followed by pages 2 and 3 and finally page 1.

If we wish to order the PageRank vector in Octave, it is enough to type the instruction

```
[psort,ranking]=sort(p,'descend')
```

The vector `psort` contains the PageRanks in decreasing order, while the vector `ranking` contains the page numbers ordered by PageRank value.

Table 4.1 PageRank and ranking for the pages of problem *A 4-page network*

Page i	PageRank p_i	Ranking
1	0.17406	4
2	0.24799	2
3	0.19324	3
4	0.38472	1

Exercises

Solve Exercise 4.3, p. 218 and Exercise 4.4, p. 219.

All Octave instructions needed for solving this problem are summarised in script `problem_google1` (see Script 4.3 at the end of the chapter). The same script can also be used to compute the PageRank of other networks, provided we substitute the above matrix A with the one associated with the new graph. □

4.8 What We have Learnt

From the Web

1. The Web is a network of gigantic dimensions (it has several billions of pages).
2. To *order a set* of objects we must associate a number with each object. The most renowned number to order webpages is the PageRank.
3. The *PageRank* of a webpage represents the probability a virtual surfer will visit that page when randomly browsing the Web.

Models

1. The Web can be modelled as a *directed graph*.
2. A directed graph is a collection of elements (called *nodes*) and connections between the nodes (called edges or *links*).
3. The essential information contained in a directed graph (the number of nodes and edges) can be stored in a matrix, called the *adjacency matrix of the graph*.
4. The mathematical model describing the movements of a virtual surfer on the Web is the system of equations $\mathbf{p} = G\mathbf{p}$, where $\mathbf{p}$ is the unknown vector of PageRanks of all the pages of the network, while G is the Google matrix associated with the network.

Numerical Methods

1. The numerical method we have chosen to solve equation $\mathbf{p} = G\mathbf{p}$ is an *iterative process*: it starts from a given initial vector $\mathbf{p}^{(0)}$ and computes an approximation $\mathbf{p}^{(k)} = G\mathbf{p}^{(k-1)}$ of the exact solution $\mathbf{p}$.
2. To stop the iterative process we must use a *stopping test*.

4.9 A Bit of History and a Glimpse Beyond

In all likelihood you have already asked yourselves why we have only spoken about Google in this chapter, and not other search engines.

The PageRank algorithm was created between 1995 and 1997 by Sergey Brin and Larry Page, then PhD students in computer science at Stanford University in California. Brin and Page realised that a search engine based on such an algorithm would be way more efficient than any other one available at the time, especially in view of the sustained growth of the Web.

During those years, to make a Web search using any one of the existing engines required a fair amount of waiting. Results were listed randomly, meaning there was no efficient criterion that organised pages in terms of their relevance in the Web. In addition, among the pages found by those erstwhile search engines one could very often find some that had nothing to do with the query. This happened because, for advertising purposes, programmers used to embed hidden words in webpages that matched the most frequent queries, for the only reason to make them show up on the lists produced by an engine.

The Brin–Page algorithm was able to overcome all these problems.

In September 1997 the two students registered the domain google.com and one year later they founded a company called Google Inc.

Clearly, after Brin and Page published their initial research, other search engines were quick to adopt similar strategies, which explains why we only know the first version of the PageRank algorithm, whereas the more complete and recent versions are kept in great secret.

The algorithm presented in this chapter goes back to the original idea of Brin and Page. In the form we have structured it, it is efficient for small networks but certainly not suitable for a network such as the present-day Web, which, as we mentioned earlier, is thought to contain over 20 billion pages.

Even without knowing the true algorithm used by Google at the moment, we may well imagine that the Google matrix G appearing in the mathematical model $\mathbf{p} = G\mathbf{p}$ is more sophisticated than what we have presented. It is reasonable to assume it is able to simulate several behavioural patterns of the virtual surfer which we did not consider.

We expect, moreover, that the iterative numerical method adopted to solve the mathematical model uses advanced acceleration techniques to reduce the number of iterations necessary to satisfy the stopping test. In fact, the lower the number of iterations, the lesser the time needed to approximate the PageRank vector.

We also believe that the initial vector $\mathbf{p}^{(0)}$ is chosen more fittingly than what we have done (assign to every page the same PageRank $1/N$), for the purpose of reducing the number of iterations necessary to satisfy the stopping test.

At last, the computing strategies, as well, will surely have been optimised over time. For instance, by exploiting the fact that G is a *sparse matrix*. A matrix is said to be sparse if the number of non-zero elements is very low compared to the overall number of elements. If we consider an average of 10 outgoing links per webpage, the number of non-zero enries in the Google matrix of a network of N pages is roughly $10N$ over a total of N^2. Consequently the percentage of non-zero elements is $10/N$. When a program uses a matrix only to compute its product with a vector (as in Algorithm 4), the zeroes can be neglected because they will not contribute to the result, and considering them only wastes computing time. In order to avoid all this there exist several strategies that only store non-zero elements, and thus save precious time during the iterations.

We can also assume that more sophisticated iterative methods are used than the one we chose for (4.9) on page 109, and probably also other stopping tests.

As always, there exist mathematical models of increasing complexity (and accuracy) and numerical models (numerical algorithms for their solution) that are

more efficient. But to show you the “basics” of the PageRank algorithm we wanted (and had) to examine very small networks and elementary algorithms.

The exploration of the Web (and social networks in general) represents one of the problems of greatest importance in the area of *discrete mathematics*. The challenges it poses are due on one hand to the exorbitant and constantly growing dimension of the associated graph. On the other hand they are caused by the need to furnish solutions at breakneck speed (to meet the increasing and frantic demands of Web surfers, i.e. us all).

4.10 List of Functions and Scripts of the Chapter

```
function G=matrix_G(A,alpha)
% matrix_G: builds the matrix G associated with the graph
% Calling instruction: G=matrix_G(A,alpha)
% Input: A = adjacency matrix of the graph
%        alpha = parameter to construct G
% Output: G = Google matrix of the network
N=length(A); % number of nodes in the network
L=zeros(N,1); S=zeros(N); % initialises L and S
for j=1:N
L(j)=sum(A(:,j));% L_j contains the total number of links
                 % from node j
end
% build S
for j=1:N
if L(j)~=0                  % if j is not a dangling node
    S(:,j)=A(:,j)/L(j);     % assignment of s_ij
else                        % otherwise if j is a dangling node
    S(:,j)=1/N;             % assignment of s_ij
end
end
% build G
G=alpha*S+(1-alpha)/N;
```

Function 4.1: `matrix_G.m`: Construction of the matrix G of a graph.

```
function [p,k]=compute_pagerank(G,epsilon,kmax)
% compute_pagerank: computes the PageRanks
% Calling instruction: [p,k]=compute_pagerank(G,epsilon,kmax)
% Input: G =  Google matrix associated with the graph
%        epsilon = tolerance for the stopping test
%        kmax =  maximum number of iterations
% Output: p =  PageRank vector
%         k = number of iterations made
N=length(G); % computes the number of nodes in the graph
p0=ones(N,1)/N; % initialises the vector p0
E=epsilon+1; % initialises the variable E for the stopping test
k=0; % initialises the iteration counter
while k< kmax && E>epsilon % iterations
    k=k+1; % increases the iteration counter
    p=G*p0; % computes p^{k}=G* p^{k-1}
    E=max(abs(p-p0)./p); % computes the error estimator E
    p0=p; % updates the vector p0 for the next step
end
```

Function 4.2: `compute_pagerank.m`: Function to compute the PageRank by Algorithm 4.

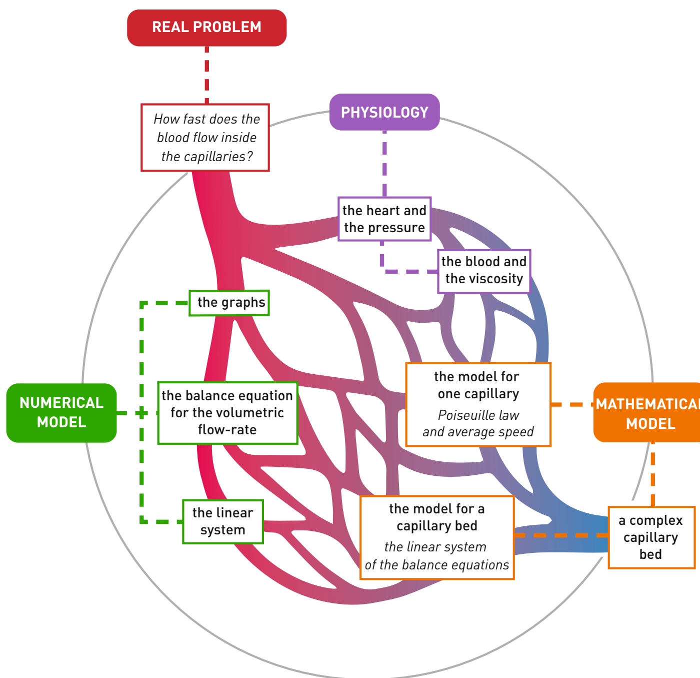
```
% script problem_google1
% solution of problem 'A 4-page network'
A=[0 0 1 0; % defines the matrix of the directed graph
   1 0 1 0;
   1 0 0 0;
   0 1 1 0];
G=matrix_G(A,0.85); % builds the Google matrix G
[p,k]=compute_pagerank(G,1e-3,100) % computes the PageRanks
[psort,ranking]=sort(p,'descend') % orders the PageRanks
```

Script 4.3: `problem_google1.m`: Script for solving exercise *A 4-page network* on page 112.

Check for updates

A Network of Capillaries

5



INGREDIENTS	Physics' notions: velocity, pressure and volumetric flow rate. Systems of linear equations and Gauss Elimination Method (GEM).
WHAT WE LEARN	Balance equations, notions of haemodynamics and fluid dynamics. The graph and its adjacency matrix, how to model a problem by means of a linear system.
PREREQUISITES	Chapter 1; Sect. 2.2 of Chap. 2. Simple one-line Octave commands (no deep knowledge required – download of package of Octave functions and scripts necessary). Section 3.1 of Chap. 3, to learn and understand the content of Octave functions and scripts.

© Springer Nature Switzerland AG 2020

A. Quarteroni, P. Gervasio, *A Primer on Mathematical Modelling*, UNITEXT 121,
https://doi.org/10.1007/978-3-030-44541-6_5

5.1 Heart, Arteries, Veins and Capillaries

The cardiovascular system allows the nutrients essential to life to reach all the cells in our body (it is estimated there are more than 37 thousand billion cells!). The fundamental engine of the system is our heart, an organ of extraordinary complexity and efficiency. The heart pumps the blood into the aorta and from there, through successive ramifications, in the bigger and smaller arteries, then in the arterioles and finally in the capillaries. And it is exactly inside the *capillary beds* that the nutrients are delivered to the cells and the toxic substances – the residues of the cellular reactions – are removed. The waste CO_2, for instance, is transported back through the venous system and is eventually released into the pulmonary alveoli where the respiration expels it.

The mathematical modelling of the entire cardiovascular system is one of the most spectacular challenges for mathematics. Here we will not attempt to address such a titanic endeavour. We shall, instead, represent a rather simple model for the blood dynamics inside the capillary beds. Whilst finding this model, we shall take the opportunity to explain certain physical properties of the dynamics of a fluid (in our case, blood), whose interest goes beyond the specific application we are dealing with at present.

Capillaries are the smallest blood vessels of the vascular system. Through their walls oxygen and nutrients are transferred from the blood to the tissues, and vice versa, carbon dioxide and catabolites pass from the tissues into the blood stream (Fig. 5.1).

In the human body there are an estimated 5 billion capillaries forming networks called *capillary beds*. Each capillary bed is supplied by a small artery (called terminal arteriole) and flows into a small vein (venule). A capillary bed is made by a variable number of elements, between 10 and 100, depending on the organ and the type of biological tissue hosting it (see Fig. 5.2).

Capillaries have an average length of half a millimetre and a diameter between 4 and 6 µm (1 µm is 10^{-6} m), roughly the size of the red cells that orderly flow through them.

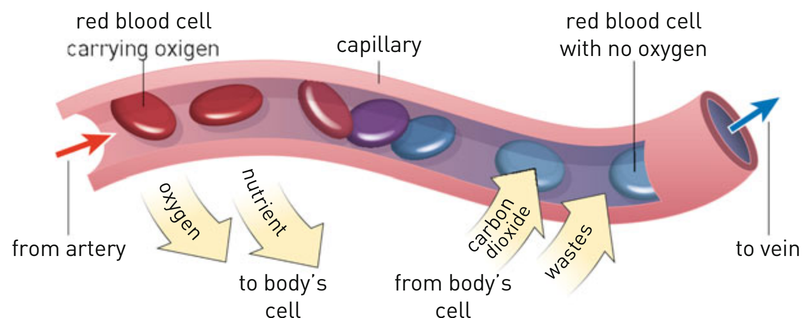


Fig. 5.1 Schematic representation of the exchange processes across a capillary wall

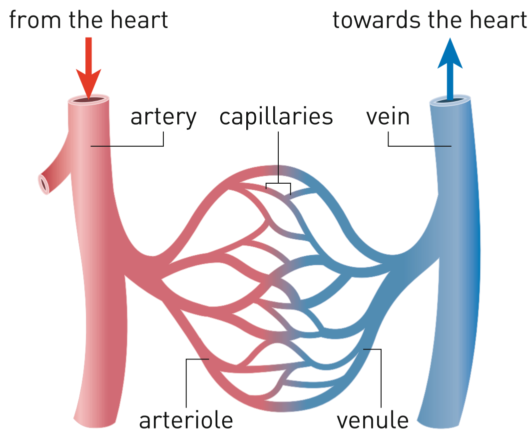


Fig. 5.2 Schematic representation of a capillary bed

Not Too Fast, Not Too Slow

The speed of blood[1] inside a capillary is typically of the order of one tenth of a millimetre per second (around 300 mm per hour), which is just about right to allow for the proper exchange of gases (oxygen and carbon dioxide), nutrients and metabolic waste. If the blood flowed too quickly, the time it took to transit through the capillaries would not be sufficient for the chemical reactions necessary for the substance exchange to happen. At the same time, though, the speed cannot go below a certain threshold, to avoid red cells from forming lumps that might obstruct the capillaries and prevent the transfer of nutrients to the tissues.

A natural question arises: what is regulating the blood flow in the capillaries? Can we tell at which speed blood flows in a capillary bed?

This problem is very complex. An accurate description of what happens in a specific capillary bed would require first of all the exact knowledge of its structure: how many capillaries it is made of, how these capillaries are organised and communicate with each other, the length of each capillary, its diameter... a mass of data that clearly is not at hand (see Fig. 5.3).

It's Easy to Say 'Blood'

Furthermore, blood is not a homogeneous fluid. It has a liquid component, called plasma, and a cellular one, made of red cells, white cells and platelets. Red cells are the delivery guys of our body and transport oxygen, nutrients and waste in the bloodstream. White cells defend our body and help fighting infections and illnesses. Platelets maintain the blood fluid in physiological conditions and help clotting in case a vessel is damaged.

It is precisely the concentration of the cellular part inside the plasma that affects the blood flow. At the same time the blood composition varies a lot: not only does it

[1] We know from physics that velocity is a vector. The speed at a point in a two- or three-dimensional channel is the modulus of the velocity vector, i.e. the square root of the sum of the squares of the components of the velocity.

Fig. 5.3 A capillary bed under the electron microscope. The white arrows point to the formation of new capillaries

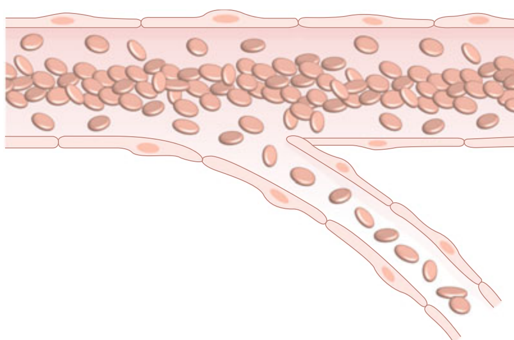


Fig. 5.4 Different concentration of red cells in an arteriole (the large vessel) and a capillary (the smaller branch)

change from person to person, but even in the same individual it varies according to the geometrical features of the blood vessels (see Fig. 5.4).

What is more, the walls of all blood vessels are not rigid but flexible, and their deformation can in turn affect the flow. Moreover, capillary walls are permeable and allow liquids to pass through.

Finally, the heart pumps intermittently at a variable rhythm, so the blood flow is not constant in time.

Writing a model capable of describing the blood flow inside the capillaries and keeping into account all these aspects is indeed very complicated, if not impossible.

Hence it becomes inevitable to introduce a number of *simplifications*, which enable us to derive a mathematical model that is nevertheless representative of the phenomenon we wish to describe and study.

To do so, we first need to learn a little *haemodynamics* (the dynamics of the blood) and, more generally, *fluid dynamics* (in general) to understand what induces the blood to flow in the vascular system, and which laws it obeys.

5.2 Understanding the Problem

5.2.1 The Beat That Keeps Us Alive

It is well known that the heart is the engine of the entire cardiovascular system. Oxygenated blood is pumped from the left ventricle into the aorta, it flows through arteries of varying dimension (from large, to medium-sized, to smaller ones) until it reaches the terminal arterioles and then the capillaries (see Fig. 5.5).

After transiting through the capillaries, where the exchange of substances with the surrounding tissues happens, blood flows into venules, and from there into larger and larger veins, it reaches the venae cavae and finally the right ventricle.

The venous blood is pushed by the right ventricle to the lungs through the pulmonary veins. Inside the pulmonary alveoli it releases into the air the toxic substances accumulated in the capillary beds, and absorbs oxygen. Then it can start its "return journey" to the heart inside the four pulmonary veins. It first enters the left atrium, then the left ventricle and through the aortic valve it is pumped into the ascending aorta and once again towards the periphery.

In normal conditions, at each heartbeat the left ventricle of an adult person pushes in the aorta about 70 ml of blood. The force exerted by the heart induces a *pressure*, denoted p and called *blood pressure*, on the walls of the vessels.

The average pressure[2] in the aorta of a healthy adult is roughly 90 mmHg.

In the systemic circulation, where blood goes from the aorta to the periphery and back to the right atrium, the average pressure decreases, as we may read in Fig. 5.6. Inside the capillaries it ranges between about 40 and 15 mmHg.

The blood propeller

The different values of the pressure at various points of the vascular system is what propels the blood to flow.

More precisely, inside any vessel the blood moves from regions of higher pressure to regions of lower pressure (Fig. 5.7). Its speed varies from vessel to vessel: it is about 40 cm/s at the beginning of the aorta, it decreases to less than 0.1 cm/s in the capillaries, and then picks up again to 10–20 cm/s in the venae cavae.

However, the different values of pressure at the extremities of a blood vessel are not the only factors regulating the flow. In this process the physical properties of blood (as we said, its composition) are of paramount importance, as well as the shape and deformability of the vessels themselves.

[2]The average pressure is computed based on the maximum high (systolic) pressure P_s and the minimum low (diastolic) pressure P_d, by the formula $p = \frac{P_s + 2P_d}{3}$. A person with systolic pressure P_s =120 mmHg and diastolic pressure P_d =80 mmHg will have $p = 93.\overline{3}$ mmHg. The millimetre of mercury (mmHg) is the unit of pressure: 1 mmHg roughly equals 133.32 pascals (Pa), and since 1 Pa=1 $\frac{\text{kg}}{\text{m s}^2} = 1 \frac{\text{g}}{\text{mm s}^2}$, we have 1 mmHg = 133.32 $\frac{\text{g}}{\text{mm s}^2}$.

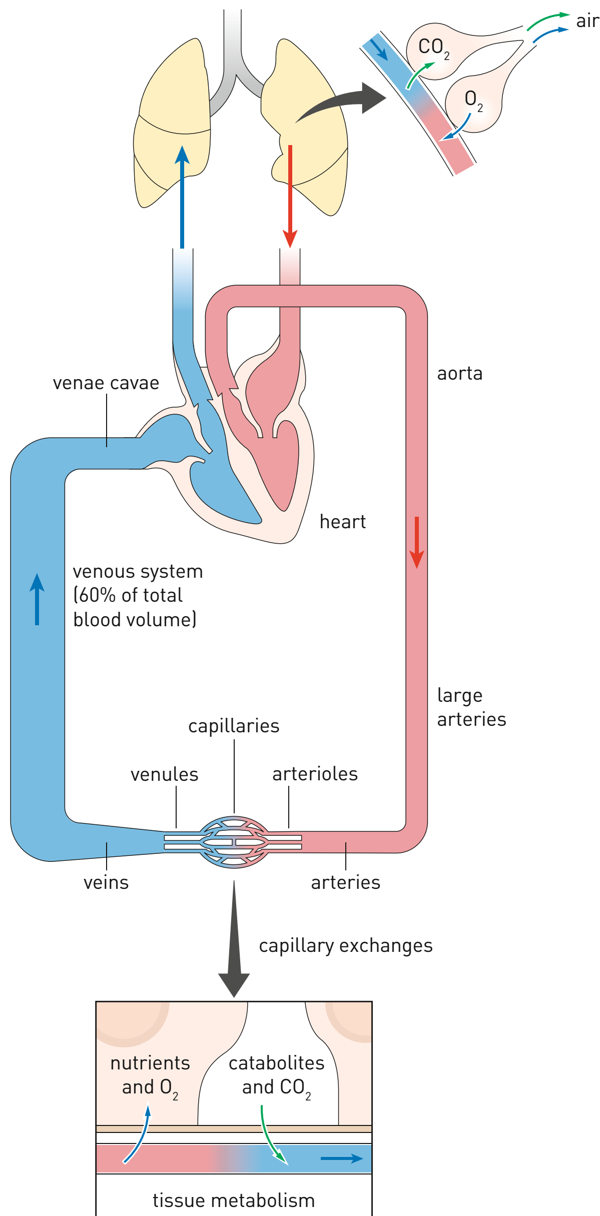


Fig. 5.5 A schematic picture of the cardiovascular system

5.2.2 Fluid Flow and Viscosity

Blood is a *viscous fluid* that in normal conditions (i.e. in a healthy person) moves under *laminar flow*.

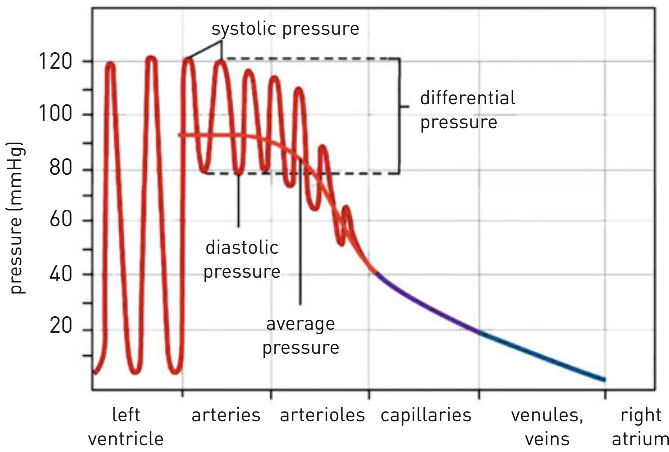


Fig. 5.6 Average pressures in the different vessels of the vascular system

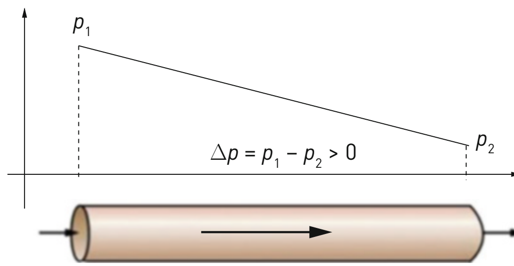


Fig. 5.7 Variation of pressure and direction of the flow: the blood flows from left to right since $p_1 > p_2$

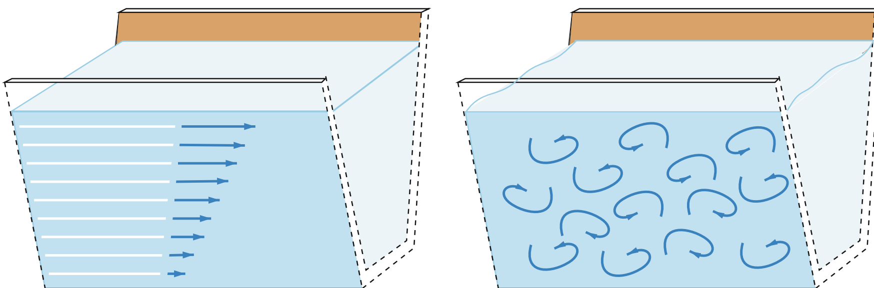


Fig. 5.8 Representation of a laminar flow (left) and a turbulent flow (right)

A *flow* is said to be *laminar* if it can be described as the motion of several layers of infinitesimal width that glide alongside one another (Fig. 5.8, left). The layers do not move at the same speed, but at different speeds.

The opposite of laminar flow is a *turbulent flow*, in which particles seem to move chaotically, as shown in Fig. 5.8, right. Moreover, there exist a whole range of behaviours (known as *regimes*) that are intermediate between pure laminar flow and utter turbulence.

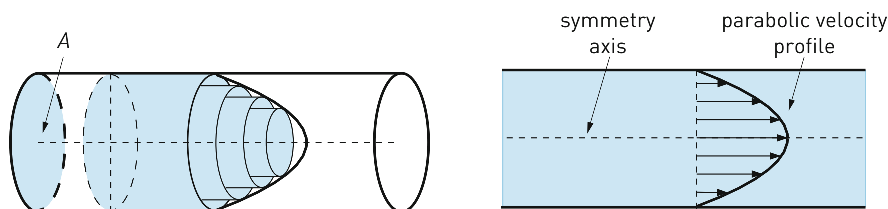


Fig. 5.9 Parabolic velocity profile of a Newtonian fluid in laminar flow inside a cylinder

The difference of speed of the layers in a laminar flow is caused by the *viscosity* of the fluid. Viscosity represents a type of inner friction (or resistance) working against the relative motion of the layers and is characteristic of each fluid, which for this reason is called a viscous fluid. When the viscosity is high it is more difficult for adjacent layers to slide along each other. Therefore to maintain a certain speed, and thus ensure a given volumetric flow rate, the difference in pressure at the extremities must be bigger.

With a little imagination we may think of viscosity as the resistance of a fluid to "spread out smoothly" on a flat surface, which is why honey is more viscous than oil, and oil is more viscous than water.

5.2.3 The Laminar Flow of a Viscous Fluid in a Cylindrical Pipe

Let us consider now a fluid in laminar flow inside a cylindrical tube (a pipe).

Our aim is describing the motion of blood in the vessels, and the pipe is a geometrical model of a vessel.

The layers in this case are not planar, but rather cylinders sliding inside one another (Fig. 5.9).

Newtonian Fluids

If the viscosity is constant (in which case one says the fluid is *Newtonian*), the speed increases from the wall of the pipe toward the central axis of symmetry (as in Fig. 5.9). The layer adherent to the wall has velocity close to zero, while along the axis the speed is maximal. Moreover, if we could cut the pipe lengthwise, we would see the particles of fluid forming a parabolic arc aligned with the symmetry axis (Fig. 5.9, right). At any point in the pipe the speed is proportional to the inverse of both the viscosity and the length of the tube. Examples of Newtonian fluids include water, oil and mercury.

Non-Newtonian Fluids

When, instead, the viscosity varies inside the fluid, meaning that the inner friction produced by adjacent layers varies, the velocity profile may not be parabolic. It may take different shapes, as for example the one in Fig. 5.10. If so, the fluid is called *non-Newtonian*. Examples of non-Newtonian fluids are blood, whipped cream, mayonnaise, toothpaste and certain varnishes.

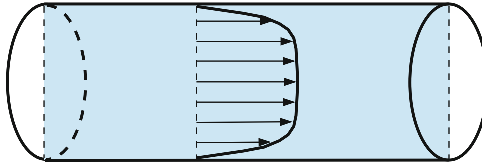


Fig. 5.10 Example of laminar flow with non-parabolic velocity profile, as exhibited by a non-Newtonian fluid

In particular, the viscosity of blood depends on the speed of the blood itself, on the radius of the blood vessel and on the concentration of red cells in the plasma. Consequently, the velocity profile in a blood vessel is not known in advance, as would happen for Newtonian fluids, and this complicates our problem further.

5.3 Simplify in Order to Model

It is clear that to describe accurately the circulation of blood in the capillary bed we should account for all the aspects seen so far: the intricate geometry of the network of vessels, walls that are not rigid, and the non-Newtonian nature of blood.

In order to proceed we shall make some simplifications to the problem. Although these assumptions will further part us from the real picture, they will still enable us to provide an acceptable representation of the phenomenon we wish to study.

Here are the simplifying assumptions we shall adopt:

1. we suppose the blood viscosity in the capillaries is constant. This will allow us to treat blood as if it were a Newtonian fluid;
2. we suppose the capillary walls are not elastic and deformable, but rigid (as a matter of fact the elasticity of the great arteries is bigger than that of the smaller ones, or of capillaries);
3. we assume each capillary is like a cylindrical pipe of length L and radius r;
4. we neglect the chemical processes underpinning the exchange of liquids through the capillary walls;
5. at last, we consider only simple capillary beds, which we may describe by networks of edges and nodes.

Under these hypotheses we can start to examine what happens inside a single capillary, then pass to simple networks made of few capillaries and eventually study a more complex network.

5.4 The Model for One Capillary

Consider a rigid cylindrical tube of length L and radius r, representing the capillary, and a Newtonian fluid, modelling blood, flowing in it under a laminar flow induced by different pressures p_1 and p_2 imposed at the extremities of the pipe, with $p_1 > p_2$.

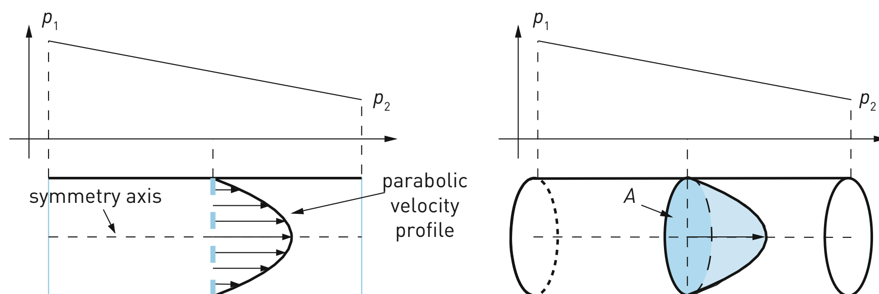


Fig. 5.11 On the left, the parabolic velocity profile (along a longitudinal section) of a Newtonian fluid inside a cylindrical pipe. On the right, we have highlighted in blue the quantity of fluid per unit of time passing through the cross-section A with velocity as in the left picture

Since the fluid is Newtonian, the velocity profile is parabolic-like, as in Fig. 5.9.

Volumetric flow rate

We call *volumetric flow rate* Q the quantity of fluid passing through a transverse cross-section per unit of time.

In dimensional terms the volumetric flow rate is therefore the ratio of volume to time, and we will measure it in $\frac{\text{mm}^3}{\text{s}}$.

The blue region in Fig. 5.11, right, contains the fluid flowing through the cross-section A per unit of time, i.e. one second. Hence its volume corresponds to the volumetric flow rate. This region is bounded by the paraboloid[3] that describes the velocity profile of the fluid (in 3 dimensions) and by the plane transversal to the pipe that intersects the paraboloid along the boundary of the tube.

Poiseuille law

The relationship between the volumetric flow rate Q and the variation of pressure $(p_1 - p_2) > 0$ is governed by the *Poiseuille law*[4]

$$Q = \frac{\pi r^4}{8\mu} \frac{(p_1 - p_2)}{L}. \tag{5.1}$$

[3] A paraboloid is a surface in space obtained by rotating a parabola about its symmetry axis. The antennas on our rooftops that we call 'parabolas' are actually portions of paraboloids.

[4] This result is attained by writing the function describing the paraboloid in Fig. 5.11 (the velocity profile of the Newtonian fluid inside the channel) and computing the volume ΔV enclosed by the paraboloid. Doing this, though, requires the use of derivatives and integrals.

The Poiseuille law establishes that the volumetric flow rate Q is proportional to the difference in pressure $(p_1 - p_2)$, to the fourth power of the radius, and to the inverse of the viscosity μ and the length L of the pipe.

By introducing the quantity

$$R = \frac{8\mu}{\pi r^4},$$

the Poiseuille law may be written more compactly as

$$Q = \frac{1}{R}\frac{p_1 - p_2}{L}.$$

The coefficient R is proportional to the viscosity which, as already mentioned, is a kind of internal friction of the fluid. To all effects, therefore, R is a *resistance*, i.e. a measure of what acts against the motion of the fluid. Note that the resistance R is proportional to the negative fourth power of the radius, whence a small reduction in the radius of the pipe will cause a big increase in resistance, and consequently a significant decrease of the volumetric flow rate Q.

The average speed

The *average speed* $\bar{v}$ of the fluid in the pipe is related to the volumetric flow rate Q by the formula

$$\overline{v} = \frac{Q}{A_c}, \tag{5.2}$$

where $A_c = \pi r^2$ is the area of the cross-section of the pipe.

The average speed represents the speed at which an *ideal fluid*[5] (one with no viscosity) would flow inside the cylinder to give the same volumetric flow rate Q of the viscous fluid under exam.

Under the assumption that a capillary can be thought of as a cylindrical tube of known length L and radius r, and assuming that the blood viscosity μ is constant and given, by the Poiseuille law (5.1) and formula (5.2) we may compute the volumetric flow rate and the average speed of blood in the capillary starting from the difference in pressure at the extremities of the capillary.

[5]Completely inviscid fluids do not exist in nature, whence the name ideal fluids. In ideal fluids the layers all move at the same speed. Air, for instance, is closer to the notion of an ideal fluid than water.

The mathematical model for the average speed

The equation

$$\overline{v} = \frac{p_1 - p_2}{R\ L\ A_c} = \frac{(p_1 - p_2)r^2}{8\mu\ L} \tag{5.3}$$

is the mathematical model we shall adopt to compute the average speed of the blood flowing with constant viscosity μ in a capillary of radius r and length L, once we know the variation of pressure $(p_1 - p_2)$ at the extremities.[6]

- The *data of the problem* are: the length L and the radius r of the capillary, the blood viscosity μ, the pressure difference between the extremities $(p_1 - p_2)$;
- the *mathematical model* is equation (5.3);
- the *solution* is the average speed $\overline{v}$.

Exercises

Solve Exercise 5.1, p. 220 and Exercise 5.2, p. 221.

5.5 Blood Flow in a Small Capillary Bed

Let us consider now a capillary bed fed by a terminal arteriole and flowing into a venule (as in Fig. 5.12, left). We want to determine the average speed of the blood inside each capillary.

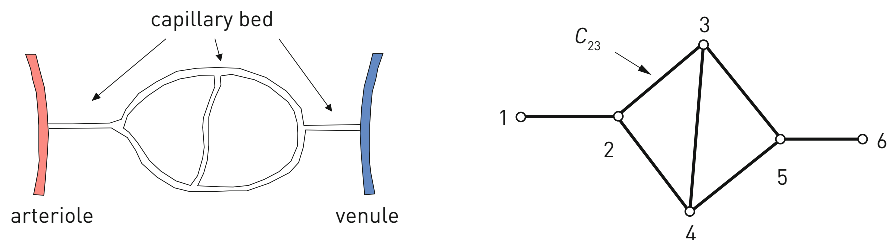


Fig. 5.12 On the left, a simple instance of a capillary bed. On the right, its representation as a network of capillaries and nodes: nodes 2 and 3 are the extremities of capillary C_{23}

[6]To check that the units of equation (5.3) are consistent, see Exercise 1.

5.5.1 The Abstraction: From the Capillary Bed to the Graph

First things first, we represent the capillary bed by a *graph* (Fig. 5.12, right), i.e. a collection of elements called *nodes* (the small white circles) connected by segments called *edges*. Each edge represents a capillary, while the nodes are the junctions between two or more vessels (capillaries, arterioles and venules). In Fig. 5.12 node number 1 identifies the point on the arteriole where the first capillary branches off, while node number 6 identifies the point where the last capillary joins the outgoing venule.

Every capillary is uniquely determined by the two nodes representing its extremities. Thus C_{12} denotes the capillary with nodes 1 and 2 as extremities, C_{23} the one between nodes 2 and 3, and more generally

$$C_{ij} \text{ indicates the capillary with extremities at nodes } i \text{ and } j.$$

We also identify C_{ji} with C_{ij}, so what we have is actually an *undirected graph*.[7]

The Adjacency Matrix of a Graph

To store the information characterising a graph, namely the links between the nodes and the lengths of the edges, we shall use a square matrix G, called the *adjacency matrix of the graph*, whose size equals the number of nodes in the graph.

Let L_{ij} be the length of edge C_{ij}. The element of the matrix G on row i and column j is denoted by g_{ij} and equals

$$g_{ij} = \begin{cases} L_{ij} & \text{if nodes } i \text{ and } j \text{ are joint by the edge } C_{ij} \\ 0 & \text{otherwise.} \end{cases}$$

Explanation: if node i and node j are connected, the element g_{ij} stores the length L_{ij} of the corresponding edge C_{ij}. If they are not connected, we set $g_{ij} = 0$.

Consider for example the graph in Fig. 5.13, where the lengths of the edges (in mm) are red. The adjacency matrix G has order $n = 6$ and takes this form

$$G = \begin{bmatrix} 0 & 0.3 & 0 & 0 & 0 & 0 \\ 0.3 & 0 & 0.3 & 0.4 & 0 & 0 \\ 0 & 0.3 & 0 & 0.5 & 0.4 & 0 \\ 0 & 0.4 & 0.5 & 0 & 0.3 & 0 \\ 0 & 0 & 0.4 & 0.3 & 0 & 0.3 \\ 0 & 0 & 0 & 0 & 0.3 & 0 \end{bmatrix}. \tag{5.4}$$

[7] In other applications (see for instance chapter 4) it is more suitable to work with *directed graphs*. These are graphs where C_{ij} represents an oriented edge, i.e. a path, from node i to node j. Hence C_{ji} and C_{ij} are distinct paths. For example, when studying the traffic on a network of roads, if C_{ij} is the road from point i to point j, it may be convenient to distinguish C_{ij} from C_{ji}, and thus account for driving directions.

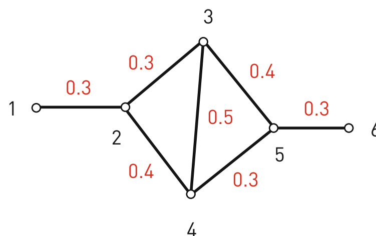


Fig. 5.13 A network of capillaries and nodes represented by a graph: in red, the lengths of the edges in mm

The first row shows the information relative to the first node: as node 1 is only connected to node 2 and the two are 0.3 mm apart, the only non-zero element on the row is $g_{12} = 0.3$.

On the second row we have the data concerning the second node: node 2 is connected to node 1, at 0.3 mm (in agreement with row 1), so $g_{21} = 0.3$. But node 2 is also connected to node 3 (at distance 0.3), so $g_{23} = 0.3$, and to node 4 (distance 0.4), so $g_{24} = 0.4$.

In a similar manner the other rows contain the information of the remaining nodes.

Finally notice that the matrix is symmetric, i.e. $g_{ij} = g_{ji}$ for every pair of indices i, j. Furthermore, $g_{ii} = 0$ for every i. The former property reflects the fact that the edges C_{ij} and C_{ji} coincide, so $g_{ij} = L_{ij} = L_{ji} = g_{ji}$. The fact that g_{ii} is zero tells that the extremities are always distinct points, i.e. we do not contemplate the presence of capillary loops, starting and ending at the same node.

5.5.2 To Compute the Speeds We Need the Pressures

We soon realise that in order to determine the volumetric flow rate and the average speed along each capillary using the Poiseuille law (5.1) and formula (5.2), we require the value of the blood pressure at all extremities, hence at all of the nodes.

Problem "Pressures in the capillary bed"

What we do have, actually, is just the blood pressure in the terminal arteriole and the venule! Put otherwise, regarding the small capillary bed under exam we only know the pressures p_1 at node 1 and p_6 at node 6. How can we find the pressures at the other nodes?

We begin by extending the Poiseuille law (5.1) to every capillary. Suppose for simplicity that the capillaries have the same radius r, so the resistance $R = (8\mu)/(\pi r^4)$ is the same for all.

Consider capillary C_{34}, of length L_{34}, and define

$$q_{34} = \frac{p_3 - p_4}{R\,L_{34}}.$$

This quantity is positive if p_3 is larger than p_4, i.e. if the fluid flows from node 3 to node 4, and is negative if the flow goes from node 4 to node 3.

More generally, let L_{ij} $(= L_{ji})$ be the length of capillary C_{ij} $(= C_{ji})$ and p_i, p_j the pressures at the extremal nodes i, j of C_{ij}. Define

$$q_{ij} = \frac{p_i - p_j}{R\,L_{ij}}, \tag{5.5}$$

a quantity that may be either positive or negative depending on the pressures p_i and p_j at the extremities.

If $p_i > p_j$, q_{ij} coincides with the volumetric flow rate of capillary C_{ij}, while if $p_i < p_j$, q_{ij} is negative and $q_{ji} = -q_{ij}$.

Since the volumetric flow rate is meant to be positive, we call *capillary flow rate* of C_{ij} the quantity $Q_{ij} = |q_{ij}|$, irrespective of the direction of the flow. Recall the volumetric flow rate is measured in $\frac{\text{mm}^3}{\text{s}}$.

5.5.3 To Compute the Pressures We Need the Balance Equations

Let us focus on one node in the capillary bed, say node 2, and look at all capillaries with an extremity at node 2: in our case, C_{21}, C_{23} and C_{24} (Fig. 5.14). Call $N_2 = \{1, 3, 4\}$ the set of nodes joint to node 2 by an edge, and let j be the generic element in N_2.

In agreement with general definition (5.5), q_{2j} is positive if $p_2 > p_j$, and so the blood flows from node 2 to node j, i.e. it “exits” node 2. If, on the contrary, q_{2j} is

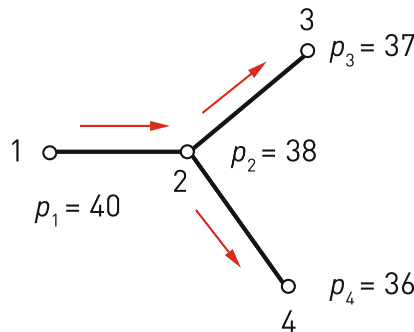


Fig. 5.14 The capillaries with an extremity at node 2. The red arrows indicate the direction of the flow dictated by the pressures, which in this picture we assume known. The blood enters node 2 through capillary C_{21} and exits through C_{23} and C_{24}

negative, because $p_2 < p_j$, the blood is flowing towards node 2 (Fig. 5.14), i.e. it "enters" node 2.

As we have assumed that the walls are impermeable and that there are no sources nor sinks at node 2 (the blood is not produced spontaneously, nor can it disappear), the sum of all terms q_{2j} must be zero:

$$q_{21} + q_{23} + q_{24} = 0.$$

The above relation is a *balance equation*, and expresses the fact that what enters node 2 must also come out.

Let us repeat this argument for a generic node i in the network. Call N_i the set of nodes joint to node i by an edge and write j for the generic element of N_i. The conclusion is the following:

Balance equation at a node

The total quantity of blood entering a node equals the total quantity of blood leaving the node. At each node i of the network the sum of all q_{ij} must be zero, where j ranges over the nodes joint to node i by a capillary.

Applying definition (5.5) for q_{21}, q_{23} and q_{24}, the balance equation at node 2 reads

$$\frac{p_2 - p_1}{R\,L_{21}} + \frac{p_2 - p_3}{R\,L_{23}} + \frac{p_2 - p_4}{R\,L_{24}} = 0.$$

Multiplying the entire equation by R (possible since $R \neq 0$) and rearranging terms with respect to the pressures $p_1, \ldots, p_4$, we obtain:

$$-\frac{1}{L_{21}} p_1 + \left(\frac{1}{L_{21}} + \frac{1}{L_{23}} + \frac{1}{L_{24}} \right) p_2 - \frac{1}{L_{23}} p_3 - \frac{1}{L_{24}} p_4 = 0.$$

Note that the coefficient of p_2 (pressure at node 2, at which we impose the balance equation) is positive and equals the sum of the reciprocals of the lengths of the capillaries reaching node 2. The coefficients of the remaining pressures p_j ($j \in N_2 = \{1, 3, 4\}$) are negative, and each one is minus the reciprocal of L_{2j}, the length of capillary C_{2j} joining nodes 2 and j.

5.5.4 A System of Balance Equations

Let us repeat the argument for nodes 3, 4 and 5, and further impose that p_1 takes the known value p_a (arteriolar pressure) and that p_6 equals p_v (venular pressure).

In this way we attain the following system of equations, whose unknowns are the pressures $p_1, p_2, \ldots, p_6$ at the nodes of the graph:

$$\begin{cases} p_1 = p_a \text{ (node 1)} \\ -\frac{1}{L_{21}}p_1 + \left(\frac{1}{L_{21}} + \frac{1}{L_{23}} + \frac{1}{L_{24}}\right)p_2 - \frac{1}{L_{23}}p_3 - \frac{1}{L_{24}}p_4 = 0 \quad \text{(node 2)} \\ -\frac{1}{L_{32}}p_2 + \left(\frac{1}{L_{32}} + \frac{1}{L_{34}} + \frac{1}{L_{35}}\right)p_3 - \frac{1}{L_{34}}p_4 - \frac{1}{L_{35}}p_5 = 0 \quad \text{(node 3)} \\ -\frac{1}{L_{42}}p_2 - \frac{1}{L_{43}}p_3 + \left(\frac{1}{L_{42}} + \frac{1}{L_{43}} + \frac{1}{L_{45}}\right)p_4 - \frac{1}{L_{45}}p_5 = 0 \quad \text{(node 4)} \\ -\frac{1}{L_{53}}p_3 - \frac{1}{L_{54}}p_4 + \left(\frac{1}{L_{53}} + \frac{1}{L_{54}} + \frac{1}{L_{56}}\right)p_5 - \frac{1}{L_{56}}p_6 = 0 \quad \text{(node 5)} \\ p_6 = p_v \text{ (node 6)} \end{cases}$$

This is a 6×6 linear system, for it is made of 6 linear equations in 6 variables (the pressures $p_1, \ldots, p_6$). It may be written in the matrix form

$$A\mathbf{p} = \mathbf{b}$$

using the notation for matrices and vectors introduced in Chap. 2. The coefficient matrix is

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ -\frac{1}{L_{21}} & \left(\frac{1}{L_{21}} + \frac{1}{L_{23}} + \frac{1}{L_{24}}\right) & -\frac{1}{L_{23}} & -\frac{1}{L_{24}} & 0 & 0 \\ 0 & -\frac{1}{L_{32}} & \left(\frac{1}{L_{32}} + \frac{1}{L_{34}} + \frac{1}{L_{35}}\right) & -\frac{1}{L_{34}} & -\frac{1}{L_{35}} & 0 \\ 0 & -\frac{1}{L_{42}} & -\frac{1}{L_{43}} & \left(\frac{1}{L_{42}} + \frac{1}{L_{43}} + \frac{1}{L_{45}}\right) & -\frac{1}{L_{45}} & 0 \\ 0 & 0 & -\frac{1}{L_{53}} & -\frac{1}{L_{54}} & \left(\frac{1}{L_{53}} + \frac{1}{L_{54}} + \frac{1}{L_{56}}\right) & -\frac{1}{L_{56}} \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

whilst

$$\mathbf{p} = \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \end{bmatrix} \quad \text{and} \quad \mathbf{b} = \begin{bmatrix} p_a \\ 0 \\ 0 \\ 0 \\ 0 \\ p_v \end{bmatrix}$$

are the solution vector and the right-hand-side vector respectively.

The mathematical model for the capillary pressures

The pressures at the nodes of a capillary bed can be computed by solving a system of linear equations of the type

$$A\mathbf{p} = \mathbf{b}.$$

This is our *mathematical model*: it was obtained by imposing the balance of the volumetric flow rates at the inner nodes of the graph associated with the capillary bed, and demanding that the pressure at the nodes connected to the arteriole and venule be known.

- The *data of the problem* are: the matrix G of the graph associated with the capillary bed (which retains the lengths of capillaries), the pressure p_a in the arteriole, the pressure p_v in the venule;
- the *mathematical model* is the linear system $A\mathbf{p} = \mathbf{b}$ built out of the balance of the volumetric flow rate at the nodes;
- the *solution* is the vector $\mathbf{p}$ of pressures at the nodes.

In Sect. 5.6 we will see how to construct the mathematical model $A\mathbf{p} = \mathbf{b}$ for a more general capillary bed.

5.5.5 From the Matrix *G* of the Graph to the Matrix *A* of the System

We begin from the matrix G of the graph associated with a capillary bed. We aim to write an algorithm that takes G and returns the matrix A of the linear system of balance equations.

To simplify our presentation we shall refer to the bed of 7 capillaries in Fig. 5.12, although the final algorithm might be used to find the pressures in more general and complex capillary beds.

Observe that the coefficient matrix A of the system, defined on page 135, has the same number of rows and columns as the adjacency matrix G of page 131. This number is the number of nodes of the graph, which we shall indicate by n. In Fig. 5.12 we clearly have $n = 6$.

The first and last rows of A are quite simple: in the top row the first element is 1 and the others are zero; in the bottom row the last element is 1 and the rest zero:

$$a_{11} = 1, \quad a_{1j} = 0 \text{ for } j = 2, \ldots, n,$$
$$a_{nn} = 1, \quad a_{nj} = 0 \text{ for } j = 1, \ldots, n-1.$$

Let us consider the intermediate rows, those labelled by $i = 2, \ldots, n-1$. Diagonal elements a_{ii} (the terms a_{ij} where $j = i$) are always non-zero, while the generic off-diagonal term a_{ij}, $j \neq i$, is non-zero only if the corresponding element g_{ij} in G is non-zero. For example, take the second row in the aforementioned G and

A: the non-zero entries in G are g_{21}, g_{23} and g_{24}, and the non-zero elements in A are a_{22} (diagonal), a_{21}, a_{23} and a_{24} (off-diagonal).

Let us now explain how to assign the correct value to the elements of A. We will again use row 2 of A, see page 135, for guidance. Its elements read

$$a_{21} = -\frac{1}{g_{21}}, \quad a_{22} = \frac{1}{g_{21}} + \frac{1}{g_{23}} + \frac{1}{g_{24}}, \quad a_{23} = -\frac{1}{g_{23}}, \quad a_{24} = -\frac{1}{g_{24}}.$$

Consider an index j varying from 1 to n, so that the elements a_{2j} may be defined by the following rule:
if $g_{2j} \neq 0$, set $a_{2j} = -\frac{1}{g_{2j}}$ and increase a_{22} by $\frac{1}{g_{2j}}$ (i.e. update a_{22} by the assignment $a_{22} = a_{22} + \frac{1}{g_{2j}}$. See Chap. 2 for assigning and updating variables in an algorithm).
Replacing the index 2 with the generic row index i $(i = 2, \ldots, n-1)$ we obtain the rule:

> let i vary from 2 to $n-1$,
> let j vary from 1 to n,
> *if* $g_{ij} \neq 0$ *then*
> set $a_{ij} = -\frac{1}{g_{ij}}$ and increase a_{ii} by the value $\frac{1}{g_{ij}}$ (i.e. update a_{ii} by the assignment $a_{ii} = a_{ii} + \frac{1}{g_{ij}}$).

Constructing the right-hand-side vector is immediate: it suffices to initialise an n-dimensional column vector $\mathbf{b}$ whose components are all zero, except the first and last ones which contain the pressures p_a and p_v in the arteriole and venule respectively.

Let us translate the above operations into an algorithm. We refer to Chap. 2 for the description of *for loops* and variable assignment, and to the following subsection for selection blocks.

Algorithm 5 *Construction of the matrix A and of the vector* $\mathbf{b}$

```
Data: matrix G of the graph, pressure values p_a, p_v
Result: matrix A and vector b
Initialise the n × n matrix A and the n-dimensional column vector b with zeroes
  everywhere;
Set a_11 = 1, a_nn = 1, b_1 = p_a and b_n = p_v;
for i = 2, ..., n − 1 do
    for j = 1, ..., n do
        if g_ij ≠ 0 then
            a_ij = −1/g_ij;
            a_ii = a_ii + 1/g_ij;
        end
    end
end
```

5.5.6 Selection Blocks in Algorithms

The instructions

if $g_{ij} \neq 0$ **then**
  $a_{ij} = -1/g_{ij}$;
  $a_{ii} = a_{ii} + 1/g_{ij}$;
end

mean the following:

if $g_{ij} \neq 0$, *do* the following operations:

- store the value $-1/g_{ij}$ in the element a_{ij} of A (the one on row i and column j),
- add to a_{ii} the value $1/g_{ij}$ (update a_{ii} by assigning $a_{ii} = a_{ii} + 1/g_{ij}$).

These constitute a *selection block* (also known as *decision block*). The instructions enclosed within **then–end** are executed only if the condition between **if** and **then** is met.

The above selection block is also known as a *selection block without alternative*, because if condition $g_{ij} \neq 0$ is not satisfied, the algorithm moves on to the next operations, namely those written after the **end** that wraps up the block.

A *selection block with alternative* is

if $g_{ij} \neq 0$ **then**
  $a_{ij} = -1/g_{ij}$;
  $a_{ii} = a_{ii} + 1/g_{ij}$;
else
  $a_{ij} = 0$
end

Here, whenever $g_{ij} \neq 0$ does not hold, the algorithm executes the alternative option, i.e. the instructions between the keywords **else–end**.

5.5.7 From the Mathematical Model to the Solution: GEM

Once all elements of A and the vector $\mathbf{b}$ have been assigned, we solve the linear system $A\mathbf{p} = \mathbf{b}$ by the Gauss Elimination Method (GEM), which was described in Chap. 2 (see Algorithms 2 and 3). To this end we may use the Octave command

```
p=A\b;
```

When typing the command \, read *backslash*, Octave calls up a function encoding GEM, as described in Algorithms 2 and 3.

Alternatively, we may call the `gem` function, which was used for exercise 3.2 (see Function 8.1, page 215) and translates in Octave language the GEM Algorithm 3 of Chap. 2. The instruction allowing to solve the linear system $A\mathbf{p} = \mathbf{b}$ with the `gem` function is

```
p=gem(A,b);
```

5.5.8 Volumetric Flow Rates and Velocities in Each Capillary

After we solve the linear system $A\mathbf{p} = \mathbf{b}$ and find the pressures at every node in the network, the volumetric flow rate Q_{ij} of capillary C_{ij} can be computed by formula

$$Q_{ij} = |q_{ij}| = \frac{|p_i - p_j|}{R\,L_{ij}} \tag{5.6}$$

(recall that we could have either $p_i > p_j$ or the opposite). Then (by formula (5.2)) the average speed $\bar{v}_{ij}$ in C_{ij} is

$$\overline{v}_{ij} = \frac{Q_{ij}}{A_c}, \tag{5.7}$$

where, let us recall, $A_c = \pi r^2$ is the area of the capillary cross-section.

Exercise: A 7-capillary bed

Consider the capillary bed of Fig. 5.12, in which the capillaries have equal radius $r = 3\mu$m and the following lengths

Capillary C_{ij}	C_{12}	C_{23}	C_{24}	C_{34}	C_{35}	C_{45}	C_{56}
L_{ij} (mm)	0.3	0.3	0.4	0.5	0.4	0.3	0.3

Suppose the blood viscosity is 2 centipoise[8] and the pressures at nodes 1 (at the arteriole) and 6 (at the venule) respectively equal $p_a = 30$ mmHg and $p_v = 20$ mmHg. Compute the pressures at the nodes, and also the volumetric flow rate and average speed in each capillary.

Solution.

- The *data of the problem* are: the blood viscosity μ, the lengths L_{ij} and the radius r of the capillaries, the arteriolar pressure p_a and the venular pressure p_v;
- the *model* is the linear system $A\mathbf{p} = \mathbf{b}$ of page 135;
- the *solution* is: the vector $\mathbf{p}$ with the blood pressures $p_1, \ldots, p_6$ at the nodes as components, the volumetric flow rates Q_{ij}, and the average speeds $\overline{v}_{ij}$ in the capillaries.

The steps:

1. define the data of the problem and construct the matrix G of the graph;

[8] 1 centipoise is 10^{-3} Pa s $= 10^{-3} \dfrac{\text{g}}{\text{mm s}}$.

2. compute the pressures at the nodes of the graph by solving the linear system $A\mathbf{p} = \mathbf{b}$;
3. compute the volumetric flow rate of every capillary with formula (5.6);
4. compute the average speed $\bar{v}_{ij}$ in each capillary by formula (5.7).

For all of this we shall use Octave.

Defining the Data of the Problem

Let us store the data of the problem (radius and lengths of the capillaries, viscosity and pressures at the extremities of the capillary bed) and build the matrix G of the graph, by calling up in Octave the predefined script

```
data_7bed
```

(see Script 5.4 on page 150).

Computing Pressures

Let us build the coefficient matrix A and the right-hand-side vector $\mathbf{b}$ by following the recipe of Algorithm 5, and then solve system $A\mathbf{p} = \mathbf{b}$ using GEM.

The Octave instructions necessary for this phase are carried out by the `pressures` function (Function 5.1 on page 148). Typing in Octave the instruction

```
p=pressures(G,pa,pv);
```

gives the following pressures:

Node	1	2	3	4	5	6
p (mmHg)	30.0000	26.8323	25.1553	24.8447	23.1677	20.0000

The variables `G, pa, pv` showing up in the instruction `p=pressures(G,pa,pv)` have been defined in script `data_7bed` (see Script 5.4 on page 150) and costitute the set of input variables for the `pressures` function. The variable `p` is the output of the `pressures` function and contains the values of the pressures that have been computed.

Figure 5.15 shows the direction of the blood flow in the capillaries: it depends on the pressures just found and keeps into account that blood flows from regions of higher pressure to regions of lower pressure.

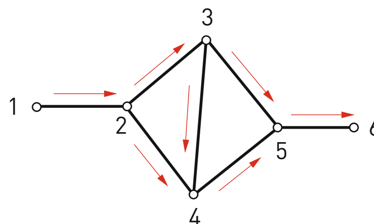


Fig. 5.15 Blood flow direction in the capillary bed under exam

After Computing the Pressures We Find the Volumetric Flow Rates

Now we shall determine the resistance R, and then use it to find the volumetric flow rates Q_{ij} in the capillaries. As the viscosity is

$$\mu = 2 \text{ centipoise} = 2 \cdot 10^{-3} \frac{\text{g}}{\text{mm s}},$$

we have

$$R = \frac{8\mu}{\pi r^4} = \frac{8 \cdot 2 \cdot 10^{-3} \frac{\text{g}}{\text{mm s}}}{\pi \cdot (3 \cdot 10^{-3}\text{mm})^4} \simeq 6.2876 \cdot 10^7 \frac{\text{g}}{\text{mm}^5 \text{ s}}.$$

Let us convert the pressure unit from mmHg to $\frac{\text{g}}{\text{mm s}^2}$, using 1 mmHg = 133.32 $\frac{\text{g}}{\text{mm s}^2}$, and apply the formula (5.6) to each capillary. The volumetric flow rates Q_{ij} (Table 5.1) may be computed using the `flow_rates` function (see Function 5.2 on page 149). More precisely with

```
Q=flow_rates(G,p,R);
```

The variables `G` and `R`, input of `flow_rates`, have been defined in the aforementioned `data_7bed` script, while the vector `p` was computed via `pressures` and contains the nodal pressures. The (output) variable `Q` is a matrix of the same order as G: its element `Q(i,j)` stores the volumetric flow rate Q_{ij} of capillary C_{ij}.

After the Volumetric Flow Rates Compute the Average Speeds

At last, using the formula (5.7) on page 139, we find the average speeds $\overline{v}_{ij}$ in the various capillaries. Also the values thus found are written in Table 5.1, and may be computed by the `average_speeds` function (see Function 5.3 on page 150). Precisely,

```
v=average_speeds(G,Q,Area);
```

The variables `G` and `Area`, input variables of `average_speeds`, have been defined in the aforementioned `data_7bed` script. The variable `Q` (another input of `average_speeds`) was found by the `flow_rates` function and encodes the volumetric flow rates. The variable `v` is a matrix of the same order of G: its element `v(i,j)` stores the average speed $\overline{v}_{ij}$ of blood in capillary C_{ij}.

Table 5.1 Volumetric flow rates and average speeds for problem *A 7-capillary bed*

Capillary	Volumetric flow rate (mm^3/s)	Average speed (mm/s)
C_{12}	$Q_{12} = 2.239 \cdot 10^{-5}$	$\overline{v}_{12} = 0.7918$
C_{23}	$Q_{23} = 1.185 \cdot 10^{-5}$	$\overline{v}_{23} = 0.4192$
C_{24}	$Q_{24} = 1.054 \cdot 10^{-5}$	$\overline{v}_{24} = 0.3726$
C_{34}	$Q_{34} = 1.317 \cdot 10^{-6}$	$\overline{v}_{34} = 0.0466$
C_{35}	$Q_{35} = 1.054 \cdot 10^{-5}$	$\overline{v}_{35} = 0.3726$
C_{45}	$Q_{45} = 1.185 \cdot 10^{-5}$	$\overline{v}_{45} = 0.4192$
C_{56}	$Q_{56} = 2.239 \cdot 10^{-5}$	$\overline{v}_{56} = 0.7918$

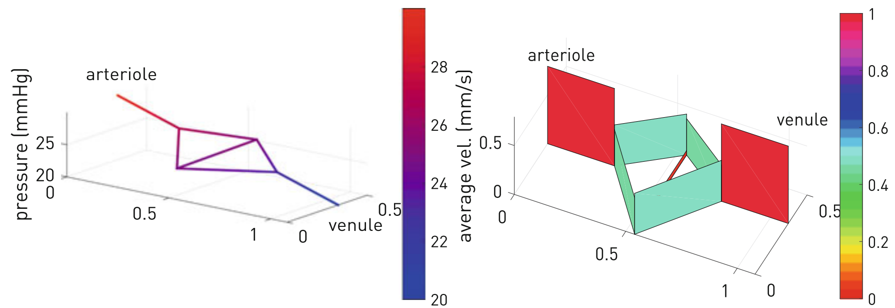


Fig. 5.16 Solutions to problem *A 7-capillary bed*. On the left, the pressures in mmHg (the nodes were joint by segments). On the right, the average speeds in mm/s (constant inside each capillary)

Note that the speeds are always positive, and the direction of the flow in the capillaries is determined by comparing the nodal pressures.

Finally, Plot the Solutions

The pressures at the nodes and the average speeds are shown in Fig. 5.16.

These pictures can be generated by executing the Octave command

```
plot_7bed(G,p,v)
```

after building the matrix `G` with the `data_7bed` script and computing the pressures `p` by the `pressures` function, the flow rates `Q` by `flow_rates` and the average speeds `v` by `average_speeds`.

Albeit present in the folder of Octave files of the book, we have not reproduced the `plot_7bed` function here. □

Exercises

Solve Exercise 5.3, p. 222.

5.6 A More Complex Capillary Bed

Let us now examine a much more complicated situation, such as the capillary bed in Fig. 5.17, left.

The Graph

Our first goal is constructing the associated graph. For simplicity we shall depict it as a small chessboard where the nodes are the vertices of the squares and the edges form the grid of perpendicular edges, see Fig. 5.17, right.

This is just a mathematical representation, however. When we draw a graph it does not really matter whether edges are straight segments or curves. What is important is that nodes and edges reflect the correct connections among the capillaries of the bed we are examining.

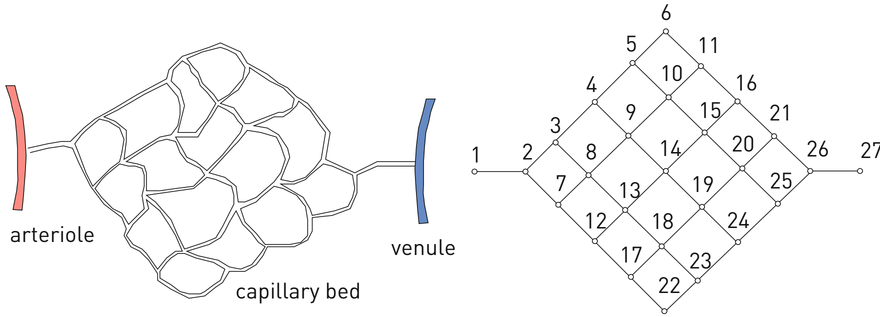


Fig. 5.17 A more complex capillary bed (left) and its representation as a network of edges and nodes (right)

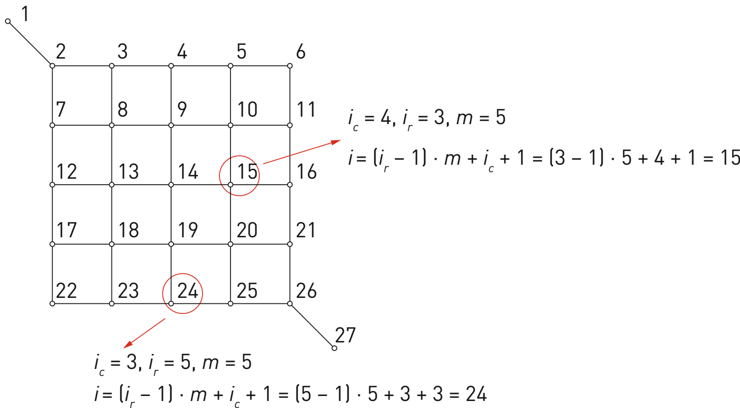


Fig. 5.18 The graph, rotated

In order to make it easier to produce the matrix G of the graph, let us rotate the graph by 45 degrees and note we have numbered nodes on each row from left to right, and rows are ordered from top to bottom (Fig. 5.18).

We have also added two fake nodes, namely the top right vertex (number 6) and the bottom left one (number 22). This merely enables us to work with the same number of nodes on each row and column and to simplify the construction of the associated adjacency matrix.

Introducing these fictitious nodes is completely irrelevant to the dynamics of the capillary bed. Consider for example node 6: it is an extremity for edges C_{56} and $C_{6,11}$ only,[9] which are consecutive capillaries. The balance equation at node 6 will guarantee that everything that flows along C_{56} will also go through $C_{6,11}$, as if node 6 were "invisible".

[9]To denote the edge joining nodes 6 and 11 we write $C_{6,11}$ separating the two numbers with a comma, and not C_{611} (which could refer to nodes 61 and 1). We will do this whenever at least one node is labelled by a multi-digit number.

We suppose for simplicity that the arcs of the graph have the same length L (in this way we are assuming that all the capillaries of the bed have length L, with the exception of those that are split in two arcs by the fake nodes 6 and 22, which have length $2L$). Call m the number of horizontal (or vertical) nodes, excluding the nodes in contact with the arteriole and the venule (in Fig. 5.18, $m = 5$).

The Adjacency Matrix of the Graph

We want to build the matrix G using an automated procedure, in terms of the number m, so that it can be used for other types of "grids" (like that in Fig. 5.17), with a different number of nodes and edges.

The matrix G for Fig. 5.17 has order $n = 27$. Node 1 is only joint to node 2, so the first row of G will consist of zeroes except for $g_{12} = L$.

To define the elements of G on the other rows, note first that diagonal elements g_{ii} are always zero since an edge links distinct nodes, i.e. there are no C_{ii} capillaries. Furthermore, as mentioned in the previous section, the matrix of the graph is symmetric, since the capillaries C_{ij} and C_{ji} are considered identical. So we need to only worry about elements g_{ij} with index j larger than i, and afterwards copy g_{ij} to the corresponding elements g_{ji} in symmetrical position with respect to the main diagonal.

Let us fix the index i of a node. The non-zero elements g_{ij} with $j > i$ are those corresponding to edges to the right and below node i.

For instance, if $i = 8$, the edges connected to node 8 are C_{38}, C_{78}, C_{89} and $C_{8,13}$. But only the last two have second index bigger than 8, namely $j = i+1 = 8+1 = 9$ and $j = i + m = 8 + 5 = 13$.

We must pay attention when, in Fig. 5.18, we consider the nodes in the right-most column and the last row. In fact if node i lies in the right-most column, it is not connected to node $i + 1$ (left-most column). Similarly, a node i in the bottom row is not connected to node $i + m$, which does not exist.

To ensure the algorithm for G deals correctly with these "peripheral" nodes, we introduce indices i_c and i_r, ranging from 1 to m, to respectively account for the columns and rows in such a way that

$$i = (i_r - 1) \cdot m + i_c + 1$$

detects the node on row i_r and column i_c (see Fig. 5.18). For example, if $i_r = 3$ and $i_c = 4$, we get $i = 15$: node 15 is exactly the one appearing on row three (from the top) and column four (from the left).

If $i_c \leq m - 1$, we set $g_{i,i+1} = L$, while if $i_r \leq m - 1$, we set $g_{i,i+m} = L$.

Finally, since node $n - 1$ (corresponding to $i_c = i_r = m$) lies both on the last row and the last column, we must treat it separately. The only other node it is connected to is number n, whence we set $g_{n-1,n} = L$.

The following algorithm generates the matrix G by executing all the operations we have explained thus far (see Chap. 2 for *for loops* and variable assignment, and Sect. 5.5.6 in this chapter for selection blocks).

Algorithm 6 *Construction of the matrix of the graph for the grid-like bed*

```
Data: m = number of nodes along one direction, L = length of arcs
Result: matrix G of the graph
set n = m^2 + 2 (total number of nodes);
initialise the n × n matrix G with zero entries;
set g_12 = L, g_21 = g_12;
for i_r = 1, ..., m (loop on the rows of the graph)
    for i_c = 1, ..., m (loop on the columns of the graph)
        set i = (i_r − 1) m + i_c + 1 (index of node on row i_r and column i_c);
        if i_c ≤ m − 1 then
            g_{i,i+1} = L (edge between a node and the node to its right);
            g_{i+1,i} = g_{i,i+1} (symmetric element);
        end
        if i_r ≤ m − 1 then
            g_{i,i+m} = L (edge between a node and the node below it);
            g_{i+m,i} = g_{i,i+m} (symmetric element);
        end
    end
end
set g_{n−1,n} = L, g_{n,n−1} = g_{n−1,n};
```

Graph → Pressures → Volumetric Flow Rates → Speeds

Once we have built the matrix G and defined all other data of the problem (arteriolar pressure p_a, venular pressure p_v, radius and length of capillaries, viscosity), the pressures, volumetric flow rates and average speeds can be determined by mimicking problem *A 7-capillary bed*, namely:

1. compute nodal pressures by solving the linear system $A\mathbf{p} = \mathbf{b}$, for which the Octave instruction is: `p=pressures(G,pa,pv);`
2. compute volumetric flow rates by formula (5.6), using the Octave instruction: `Q=flow_rates(G,p,R);`
3. compute average speeds by formula (5.7), using the Octave instruction: `v=average_speeds(G,Q,Area);`

Exercise: A grid-like bed

Consider the capillary bed of Fig. 5.17, whose capillaries all have radius $r = 2.5$µm and length $L = 0.5$ mm (except for the uppermost and lowest ones in Fig. 5.17, left, whose length is $2L$). Suppose the blood viscosity is 1.9 centipoise and the pressures at nodes 1 (joint to the arteriole) and 27 (joint to the venule) are $p_a = 40$ mmHg and $p_v = 15$ mmHg. Compute the pressures at the nodes of the capillary bed, plus the volumetric flow rate and the average speed in each capillary.

Solution.

- The *data of the problem* are: the viscosity μ of blood, the length L and radius r of the capillaries, the blood pressures p_a in the arteriole and p_v in the venule;

- the *model* is the linear system obtained by imposing the balance equations for the volumetric flow rate at nodes 2, 3,..., 26 together with the conditions $p_1 = p_a$ and $p_{27} = p_v$.
- the *solution* is given by the values $p_1, \ldots, p_{27}$ of the blood pressure at the nodes, the volumetric flow rates Q_{ij} and the average speeds $\overline{v}_{ij}$.

We have prepared the script `data_grid_like_bed` (Script 5.5, page 151) to define the data of the problem and generate the matrix G according to the instructions of Algorithm 6. To execute it just type the Octave command

```
data_grid_like_bed
```

As for the pressures, volumetric flow rates and speeds, we may call up the same functions used for problem *A 7-capillary bed*. Typing the instructions:

```
p=pressures(G,pa,pv);
Q=flow_rates(G,p,R);
v=average_speeds(G,Q,Area);
```

produces on screen the pressures at the 27 nodes, the volumetric flow rates and the average speeds. Instead of writing down the values obtained, we plotted pressures and average speeds in Fig. 5.19, which may be generated in Octave by the instruction:

```
plot_grid_like_bed(G,p,v)
```

The `plot_grid_like_bed` function may be found in the Octave folder and was therefore not reproduced here.

Note how the pressure variation is bigger in those capillaries that are closer to the arteriole and venule, while it is distinctly lower in the other capillaries.

The speeds are proportional to the pressure variations: a bigger change in pressure at the extremities of a capillary will result in a higher speed. Values vary between a minimum 0.08 mm/s in capillaries close to nodes 6 and 22 (the farthest from the arteriole and venule) to a maximum 0.66 mm/s, closest to the arteriole and venule. □

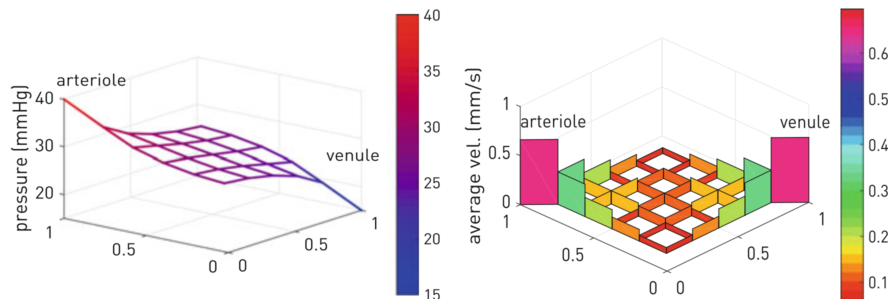


Fig. 5.19 The solutions to problem *A grid-like bed*. Above, the pressures in mmHg (the values at nodes are connected by segments). Below, the average speeds in mm/s (constant inside each capillary)

Exercises

Solve execise 5.4, p. 224.

5.7 What We Have Learnt

Haemodynamics and Fluid Dynamics

1. Blood can be assimilated to a *non-Newtonian viscous fluid*, and in physiological conditions blood flow is laminar;
2. blood vessels are not rigid, but elastic, pipes;
3. blood flows from higher-pressure regions to lower-pressure regions. There is a resistance opposing the flow that depends on the viscosity (which in turn is a function of the speed), the radius and length of the blood vessel;
4. the velocity profile of a Newtonian fluid inside a cylindrical pipe is *parabolic*;
5. the *Poiseuille law* encodes the relationship between the volumetric flow rate in a rigid cylindrical pipe and the difference in pressure at the extremities of the pipe of a Newtonian viscous fluid in laminar flow;
6. at each node in a network of capillaries a *balance equation* holds: what flows into a node equals what flows out.

Models

1. From the geometrical point of view capillaries are described as rigid cylindrical pipes;
2. a capillary bed is modelled as a *graph*;
3. blood is a Newtonian fluid;
4. the mathematical model of the system of balance equations at the nodes of the capillary bed is a *system of linear equations*.

Mathematical Tools

1. A *graph* is a collection of *nodes* and *edges* joining the nodes. The essential information conveyed by a graph may be stored in a matrix called the adjacency matrix of the graph.
2. The Gauss Elimination Method (GEM) is a technique for solving linear systems. We introduced it in Chap. 2, and here we have seen it in action.

5.8 What Lies Beyond

Well, there is no need to say there still are a thousand questions one could ask.

How do we modify our model if we want to consider a person exercising? To answer we should modify the response model of the capillary walls, and introduce an equation that accounts for how walls react to the stress induced by higher blood pressure and bigger volumetric flow rate.

Or how do we take into account a different blood viscosity, caused, say, by a reduced red-cell volume (as in presence of anaemias, see Fig. 5.20)? In such a case we would likely need three-dimensional models, which would make the mathematics much more complicated.

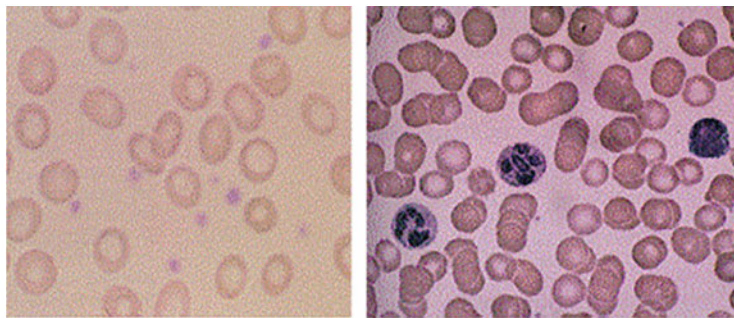


Fig. 5.20 Anaemic blood (left) and normal blood (right)

Or still, what should we do if we wanted to consider the dynamics of a drug, whose concentration varies in space and time in the various meanders of our capillary bed? Here, too, we would need a model that couples the blood dynamics (as seen in this chapter) with the dynamics of the drug (which is clearly affected by the former).

To sum up, it is patent that the more we want to know, the more accurate and reliable we should be making our mathematical models. The payoff will be the possibility to provide physicians with increasingly meaningful answers (both qualitatively and quantitatively) to support diagnoses and treatments.

5.9 List of Functions and Scripts of the Chapter

```
function [p]=pressures(G,pa,pv)
% pressures: computes the pressures in a capillary bed
% Calling instruction: [p]=pressures(G,pa,pv)
% Input:  G = matrix of the graph
%         pa, pv = pressures in the arteriole and venule
% Output: p = vector of pressures at the nodes of the bed

% Build the matrix A ...
n=length(G); A=zeros(n); A(1,1)=1; A(n,n)=1;
for i=2:n-1
    for j=1:n
        if G(i,j)~=0
        A(i,j)=-1/G(i,j);
        A(i,i)=A(i,i)+1/G(i,j);
        end
    end
end
b=zeros(n,1);  % ... and the right-hand-side term
b(1)=pa; b(n)=pv;
p=A\b; % solves the linear system by GEM
for i=1:n  % prints the pressures
fprintf('The pressure at node %d equals %8.4f mmHg\n',i,p(i))
end
```

Function 5.1: `pressures.m`: function for computing pressures in a capillary bed.

```
function [Q]=flow_rates(G,p,R)
% flow_rates: computes the volumetric flow rates in a capillary
% bed
% Calling instruction: [Q]=flow_rates(G,p,R)
% Input:  G = matrix of the graph (lengths of edges in mm)
%   p = vector of pressures at the nodes of the bed (in mmHg)
%   R = resistance in capillaries (in g/(mm^5 s) )
% Output: Q = volumetric-flow-rate matrix (in mm^3/s)

n=length(p); % determines the number of nodes
Q=zeros(n);  % initialises the volumetric-flow-rate matrix
% compute volumetric flow rates and print
for i=1:n  % loop over nodes
  for j=1:n  % loop over nodes
    if G(i,j)~=0
% if nodes i and j are connected, compute the volumetric
% flow rate Q_ij
      Q(i,j)=133.32*abs(p(i)-p(j))/(R*G(i,j));
% print the volumetric flow rates (recall C_ij=C_ji,
% so it suffices to print the values for j>i)
      if j>i
        fprintf('The flow rate C_%1d%1d is %8.3e mm^3/s\n',...
              i,j,Q(i,j))
      end
    end
  end
end
```

Function 5.2: `flow_rates.m`: function for computing volumetric flow rates in a capillary bed.

```
function [v]=average_speeds(G,Q,Area)
% average_speeds: computes the average speeds for a capillary
% bed
% Calling instruction: [v]=average_speeds(G,Q,Area)
% Input:  G = matrix of the graph  (lengths of edges in mm)
%         Q = matrix of volumetric flow rates (in mm^3/s)
%         Area = area of the capillary cross-section (in mm^2)
% Output: v = matrix of average speeds (in mm/s)

n=length(G);  % determines the number of nodes
v=zeros(n); % initialises the matrix of speeds
for i=1:n
 for j=1:n
  if G(i,j)~=0
% if nodes i and j are connected, compute the average speed
% in capillary C_ij
   v(i,j)=Q(i,j)/Area;
% print the average speeds (recall C_ij=C_ji, so it suffices to
% print the values for j>i)
   if j>i
    fprintf('The average speed at C_%1d%1d is %8.4f mm/s\n',...
       i,j,v(i,j))
   end
  end
 end
end
```

Function 5.3: average_speeds.m: function for computing average speeds in a capillary bed.

```
% script data_7bed
% this script defines the data and the matrix for the graph of
% problem ''A 7-capillary bed''
r=3e-3; % radius of capillaries (mm)
mu=2e-3; % blood viscosity g/(mm*s)
pa=30; pv=20; % pressure in the arteriole and venule (mmHg)
R=8*mu/(pi*r^4); % resistance in capillaries g/(mm^5*s)
Area=pi*r^2; % area of the capillary cross-section (mm^2)
% G = matrix of the graph encoding capillary lengths
G=[0   0.3 0   0   0   0;
   0.3 0   0.3 0.4 0   0;
   0   0.3 0   0.5 0.4 0;
   0   0.4 0.5 0   0.3 0;
   0   0   0.4 0.3 0   0.3;
   0   0   0   0   0.3 0];
disp('The data of problem "A 7-capillary bed" are defined')
```

Script 5.4: data_7bed.m: script to define data and graph matrix for problem *A 7-capillary bed*.

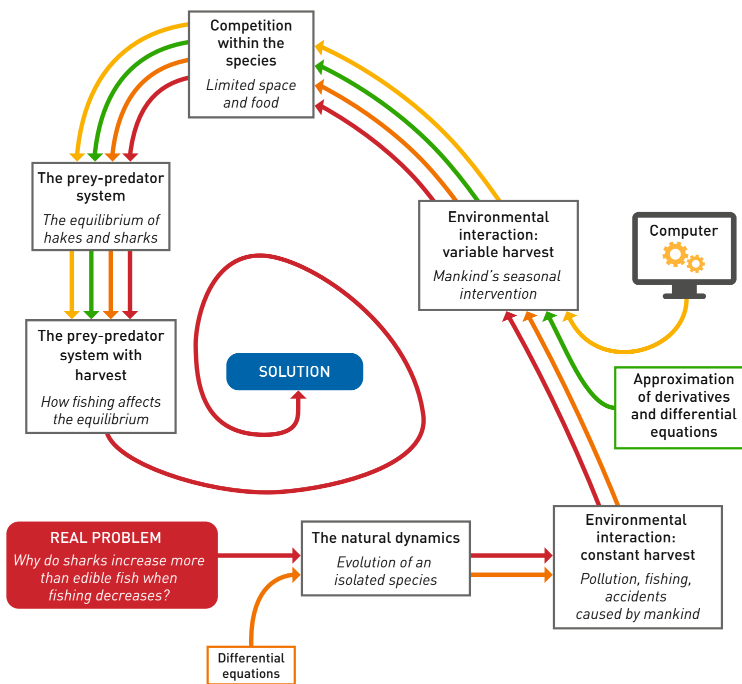
```
% script data_grid_like_bed
% this script defines the data and matrix of the graph for
% problem "A grid-like bed"
L=0.5; % capillary length (mm)
r=2.5e-3; % capillary radius (mm)
mu=1.9e-3; % blood viscosity g/(mm*s)
pa=40; pv=15; % pressure in the arteriole and venule (mmHg)
R=8*mu/(pi*r^4); % resistance in the capillaries g/(mm^5*s)
Area=pi*r^2; % area of capillary cross-section
if ~(exist('nc','var')) % checks that nc is defined
  nc=4; % number of capillaries along each direction of the
        % graph
end
m=nc+1; % m = number of nodes along each direction
% G = matrix of the graph encoding capillary lengths
n=m*m+2;  G=zeros(n);
G(1,2)=L; G(2,1)=G(1,2);
for ir=1:m
    for ic=1:m
        i=(ir-1)*m+ic+1;
    if ic <= m-1
        G(i,i+1)=L; G(i+1,i)=G(i,i+1);
    end
    if ir <= m-1
        G(i,i+m)=L; G(i+m,i)=G(i,i+m);
    end
    end
end
G(n-1,n)=L; G(n,n-1)=G(n-1,n);
disp('The data of problem "A grid-like bed" have been defined')
```

Script 5.5: `data_grid_like_bed.m`: script to define data and graph matrix for problem *A grid-like bed*.

Check for updates

Predators and Preys in the Maths Ocean

6



INGREDIENTS	Functions, derivatives.
WHAT WE LEARN	Definitions of differential equation and Cauchy problem. Methods for approximating derivatives. Methods for solving ordinary differential equations.
PREREQUISITES	Chapter 1. Ability to use Octave at the level of simple one-line commands (deeper knowledge not required – download of package of Octave functions and scripts necessary). Section 3.1 of Chap. 3 to understand and learn the content of Octave functions and scripts.

© Springer Nature Switzerland AG 2020

A. Quarteroni, P. Gervasio, *A Primer on Mathematical Modelling*, UNITEXT 121,
https://doi.org/10.1007/978-3-030-44541-6_6

6.1 Fishing in the Adriatic Sea

Around 1925 the Italian biologist and natural historian Umberto D'Ancona[1] became interested in fish populations and problems related to fishing activities in the northern Adriatic Sea. He had observed that between 1915 and 1918 the big reduction in fishing caused by the First World War had altered the natural equilibrium within the fish fauna. This had fostered the spread of sharks, rays and other predatory species over their prey, the fish living on plankton and small invertebrates.

The statistical data available to D'Ancona was the percentage of predatory fish caught, over the entire catch, in the waters around Fiume (today's Rijeka, Croatia) between 1914–1923. More precisely, D'Ancona knew the annual average of said percentage, shown in Fig. 6.1.

D'Ancona problem

It was not clear why the lower catch during the war had caused the population of predators to grow so much, without any increase of prey species (which, by the way, were the type of edible fish the market demanded most).

Here is an example of a problem of concrete interest that allows for a mathematical formulation.

The *complexity* of the problem, however, is not trivial. First of all, there exist several species of fish that interact with one another, with the environment and with mankind. There also are a number of variables at play: climate conditions, the quantity of food available to each species (some species eat plankton, others

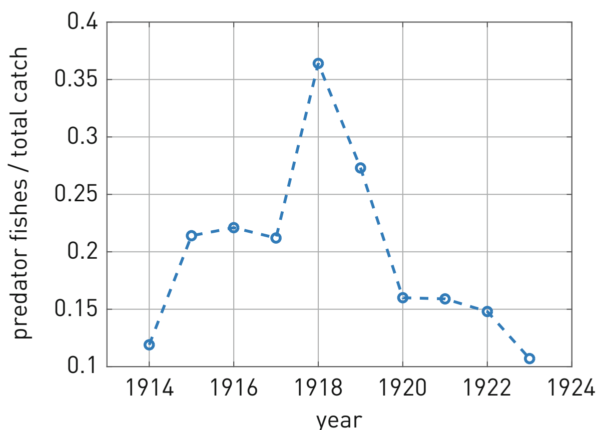


Fig. 6.1 The statistical data on which the work of Umberto D'Ancona is based, i.e. the percentage of predatory fish to the whole catch in the northern Adriatic between 1914–1923

[1] Umberto D'Ancona (1896–1964).

feed on other fish), the reproductive speed of the species, their longevity, human involvement through fishing…

We will have to introduce certain *simplifications* that will enable us to infer a mathematical model that we can handle.

Once we manage to *model* and *solve* the problem mathematically, we will need to check that the results obtained agree with the statistical data and the measurements provided by the real scenarios. The latter is the *validation* phase of the process: if the answer is yes, we shall accept the validity of the mathematical model.

After that, we may want to *enhance the model*, that is remove one or more simplifying assumptions, in order to get closer to the real problem and provide a better mathematical solution.

6.2 Simplify in Order to Model

In order to construct a mathematical model we often need to simplify the problem. Although these simplifications might lead us farther away from the faithful description of reality, they nevertheless allow us to provide an acceptable representation of it.[2]

Fig. 6.2 *Vito Volterra* (1860–1940) obtained his Physics degree in 1882 with a thesis on hydrodynamics, and became Professor of Rational Mechanics at the University of Pisa at the age of 23. Early on he shifted his research interests towards mathematics and made fundamental contributions to the theory of functions and differential equations. His name is mostly associated with the study of population dynamics, see `http://volterra.uniroma2.it` (Photo: The Burndy Library, Cambridge, Massachusetts)

[2] We will take inspiration from the procedure the famous mathematician Vito Volterra published between 1920 and 1930 (see Fig. 6.2), in particular in the article *Variazioni e fluttuazione del numero d'individui in specie animali conviventi*, Memorie del R. Comitato talassografico italiano, Mem. CXXXI, 1927 (in *Opere Matematiche* di Vito Volterra, vol. quinto 1926–1940. Accademia Nazionale dei Lincei, Roma, 1962).

(a) Photo by Jet Kim from Unsplash, 2019

(b) Photo by Giustiliano Calgaro from Pixabay, 2007; id 1302291

Fig. 6.3 Homogeneous fish population (**a**); Predator and preys (**b**)

In other words we should isolate the action we wish to study from less influential, and hence neglectable, aspects.

Which are the most interesting aspects? What guides us in neglecting certain facets of reality?

The simplifying hypotheses we shall introduce are suggested by the need to build a model within the reach of the mathematical and computational tools at our disposal.

For the case at hand, the interesting aspects are the interaction between predatory species and prey species, and the influence exerted by fishing on the proliferation of the various species (see Fig. 6.3).

Based on all this, we make the following simplifications.

1. We assume that the dynamics of fish populations does not depend on environmental factors: fish constitute an *isolated system* from the rest of the marine world.
2. Although there exists several types of predatory fish and several of prey fish, we separate them in *just two populations*.
3. We suppose each one of the populations is *homogeneous*: all prey fish behave in the same way, and so do predators. In either population we do not distinguish adults from juvenile individuals. The reality is clearly different, since fingerling preys are more likely to be eaten and hence they are more vulnerable. At the same time, though, juvenile fish might be considered more shielded, as they can hide in nooks and crannies more easily than adults.
4. Predators are supposed to feed only on the prey species (which is in limited supply) and preys only live on planktoni.
5. Finally, we assume that any form of fishing affects the two populations equally, so that the average catch is split halfway between preys and predators.

We shall thoroughly explain that the mathematical model obtained under all these hypotheses is a *system of two differential equations*.[3]

[3]A differential equation is an equation encoding the relationship between a function and its variations (derivatives).

6.3 Understanding the Problem

Let us begin from the data of Fig. 6.1: the numbers give, for every year from 1914 to 1923, the percentage of predator fish over the entire catch, and implicitly assume that the sizes (also called *abundances* by biologists) of both predator and prey populations throughout the years are known.

On the x-axis we place time, on the y-axis a percentage, written as a decimal number, relative to the two population sizes in time: in other words we have a *function of time*.

So let us introduce the independent variable t, representing time, and the two functions $p(t)$, $s(t)$ describing the size, depending on time, of prey and predator populations respectively (the letter s stands for 'sharks').

If we suppose the proportion of preys to predators within the fishery stock is the same as that in the entire sea, the data of Fig. 6.1 represent the annual average of the function

$$R(t) = \frac{s(t)}{p(t) + s(t)},$$

i.e. the ratio of predators to the entire fish population in the sea.

Mathematical modelling

We are interested in building a model that, starting from a known situation at some given time t_0 (i.e. an estimate of how many preys and predators are present in the sea at time t_0), is capable of describing the evolution of the sizes $p(t)$ and $s(t)$ of the two populations.

To construct this model we must translate into mathematical relationships the interactions between the two populations and the effect fishing has on them.

Once $p(t)$, $s(t)$ are known, we can evaluate the function $R(t)$ and check whether in the model the values of $R(t)$ increase as fishing decreases, as in the real world. This comparison will allow us to validate, or not, our model.

Before we formulate the model for the study of the populations, we must take a step back and understand how to model the phenomena governing the evolution of one population alone.

Let us therefore isolate the entity we wish to describe, ignore the surrounding conditions and proceed by the aforementioned three phases:

1. simplify in order to model;
2. validate the model;
3. enhance the model to attain a more realistic description.

6.4 The Dynamics of One Population

In this section we will try to understand how *a single population* evolves. We will first consider the simplest situation and study the natural dynamics of the population. We shall, in other words, suppose the population has unlimited food resources and there are no predators to threaten it.

Later we will enhance the model by adding to the dynamics the interaction with the environment, such as fishing for example (from page 162). Eventually we will make the model more realistic by examining the situation in which the members of the population start competing because of food shortages or lack of living space (page 179).

Along the way we will introduce new mathematical concepts (such as differential equations), and simple numerical methods to approximate derivatives (page 165) and solve differential equations (page 171).

6.4.1 The Natural Dynamics

Consider a population with unlimited food resources and no predators. Its size can vary only due to births and natural deaths caused by characteristic factors, called birth rate n and death rate m. We will suppose n, m are constant throughout the period of interest.

The *birth rate* n is the ratio of newborns per unit of time (a year, say) to the average size of the population over the same time. The *death rate* m is similarly the ratio of natural deaths per unit of time to the average size of the population over the same time.

Both n and m are real numbers lying in the interval $[0, 1]$, and we may think of them as (constant) natality and mortality rates.

For example $n = 0.02$ means the population has grown 2% due to births, while $m = 0.01$ indicates the population has shrunk due to natural deaths by 1%.

The *population change* y at time t is therefore the difference between the number of births and natural deaths

$$n\,y(t) - m\,y(t) = (n - m)y(t).$$

If we define

$$r = n - m,$$

called the *net growth rate*, then the variation of population y at time t equals $r\,y(t)$. Since n, m are real numbers in $[0, 1]$, we have $-1 \leq r \leq 1$.

Infinitesimal calculus warrants that the variation of a quantity $y(t)$ can be expressed in terms of its first derivative $y'(t)$.

The mathematical model of Malthus

The law that models, in time, the size of a population with unlimited resources and no predators is the *law of exponential evolution* (also known as *Malthus law*[4])

$$y'(t) = r\, y(t), \tag{6.1}$$

where r is the net growth rate.

When $r > 0$, i.e. when the birth rate n is larger than the death rate m, it is rather intuitive to conclude that the population will continue to grow in time. When $r < 0$, instead, the population will tend to extinction.

Differential equations

Equations such as (6.1), where the unknown is a function and also the derivative of said function is present, are called *differential equations*.

Examples of functions $y(t)$ solving (6.1) are the exponential functions

$$y(t) = e^{rt}.$$

Two of these, corresponding to $r = 0.5$ and $r = -0.5$, are shown in Fig. 6.4.

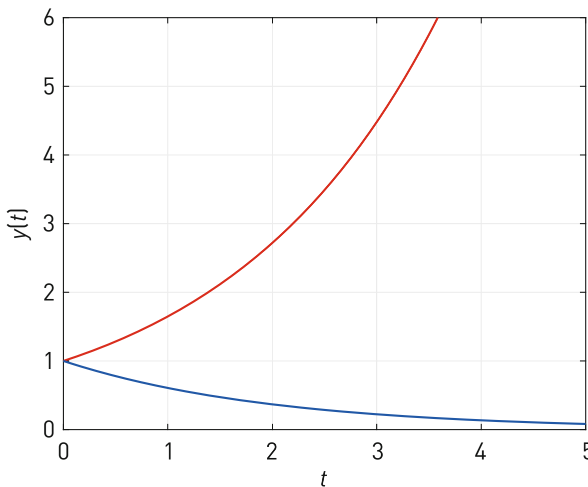


Fig. 6.4 Two exponential functions of type $y(t) = e^{rt}$ that satisfy Malthus evolution law (6.1). In red the function with $r = 0.5$, in blue the one with $r = -0.5$

[4]Thomas Robert Malthus (1766–1834) was an English economist and demographer.

Note that all exponential functions of the type (c is an arbitrary number)

$$y(t) = c\,e^{rt}$$

satisfy equation (6.1). In fact, by computing the derivative of $y(t)$ and recalling that t is the independent variable while r, c are constant, we obtain

$$y'(t) = r\,\underbrace{c\,e^{rt}}_{y(t)},$$

precisely the relation given by (6.1).

As c can be any real number, we deduce that there exist infinitely many functions $y(t)$ satisfying (6.1). In other terms, the differential equation *admits infinitely many solutions*.

We can characterise uniquely *the* solution of the differential equation if we know the size y_0 of the population at some time t_0. This information, expressed by

$$y(t_0) = y_0,$$

is called *initial condition*.

The mathematical model of the natural dynamics

The differential problem[5]

$$\begin{cases} y'(t) = r\,y(t) & \text{if } t \geq t_0 \\ y(t_0) = y_0 \end{cases} \tag{6.2}$$

is a mathematical model for the natural dynamics of a population. In other words it describes the evolution of a population with given growth rate r, unlimited food resources and no interactions with its surroundings, starting from a given instant t_0 at which the population size is known.

The solution to (6.2) is

$$y(t) = y_0\,e^{r(t-t_0)},$$

hence an increasing exponential if $r > 0$, and a decreasing exponential if $r < 0$. Both functions in Fig. 6.4 satisfy the condition $y(t_0) = y_0$ with $t_0 = 0$ and $y_0 = 1$.

To check that $y(t) = y_0\,e^{r(t-t_0)}$ is indeed a solution to Cauchy problem (6.2), we look at the first derivative:

$$y'(t) = y_0\,r\,e^{r(t-t_0)} = r\,\underbrace{y_0 e^{r(t-t_0)}}_{y(t)} = r\,y(t),$$

[5]This is an instance of a *first-order Cauchy problem*.

Fig. 6.5 Dolphins. Photo by Wolfgang Zimmel from Pixabay, 2017; id 2691864

which confirms $y(t)$ satisfies the differential equation $y'(t) = ry(t)$ of the Cauchy problem.

Next we compute $y(t_0)$:

$$y(t_0) = y_0 e^{r(t_0 - t_0)} = y_0 e^0 = y_0,$$

whence $y(t)$ satisfies the initial condition of Cauchy problem (6.2) as well.

Exercise: Aeolian dolphins

It is estimated that the population of dolphins in the waters around the Aeolian Islands, southern Italy, in July 2017 was down to 100 individuals. Suppose dolphins have unlimited food, that their natural predators are absent, that the net growth rate is $r = 0.06$, and there are no interactions with the environment (no dolphin leaves its pod, no newcomers join, and no human intervention). Determine how many dolphins are expected to populate the Aeolian Sea by July 2025. (See Fig. 6.5.)

Solution. Call $y(t)$ the function expressing the number of dolphins, and let us fix the unit of time to be one year, so that July 2017 corresponds to $t_0 = 2017.5$. Since the population does not interact with other species, the evolution of our system is given by model (6.2) with $r = 0.06$ and $y_0 = 100$.

- The *data* are $r = 0.06$, $t_0 = 2017.5$, $y_0 = 100$;
- the *mathematical model* is the Cauchy problem (6.2) of page 160;
- the *solution* is the value assumed by the exponential function that solves the Cauchy problem when $t = 2025.5$.

Note that we could take the initial time 2017.5 as time zero. In that case the final time 2025.5 would become 8, but all the rest would stay the same. Substituting our data in model (6.2) gives the Cauchy problem

$$\begin{cases} y'(t) = 0.06\, y(t), & \text{if } t \geq 2017.5 \\ y(2017.5) = 100, \end{cases}$$

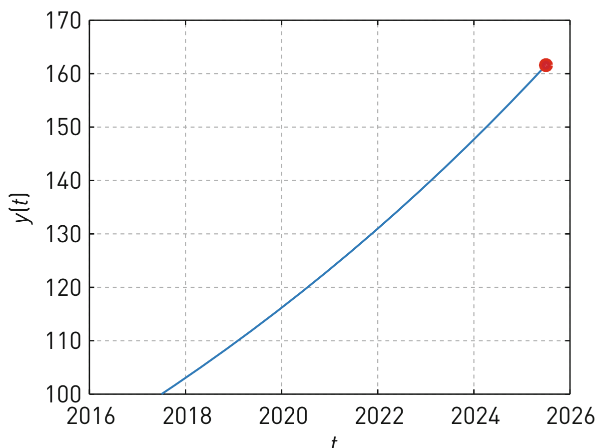


Fig. 6.6 The number of dolphins between July 2017 and July 2025, solution to problem *Aeolian dolphins*

whose solution is the exponential function

$$y(t) = 100\, e^{0.06(t-2017.5)}.$$

So to find the number of dolphins in July 2025 it is enough to evaluate y at $t = 2025.5$:

$$y(2025.5) = 100\, e^{0.06(2025.5-2017.5)} = 100\, e^{0.48} \simeq 161.60.$$

Figure 6.6 shows the function $y(t)$ expressing the number of dolphins in the Aeolian Sea from July 2017 to July 2025. □

Exercises

Solve Exercise 6.1, p. 226.

6.4.2 The Interaction with the Environment: Constant Harvest in Time

Let us make the model a little more realistic by supposing the population size under exam can also vary because of some interaction with its environment (we are thus getting closer to the real-world situation). We can for instance think that in a unit of time (one year) a certain quantity $B > 0$ of individuals is caught (or is eaten by natural predators, or falls victim to human-caused accidents).

The law of exponential evolution (6.1) must be replaced by another equation that accounts for the harvest, namely

$$y'(t) = r\, y(t) - B.$$

Still assuming the initial condition $y(t_0) = y_0$ holds, the new mathematical model describing the dynamics of a population subject to constant harvest is

$$\begin{cases} y'(t) = r\,y(t) - B, & \text{if } t \geq t_0 \\ y(t_0) = y_0 \end{cases}. \tag{6.3}$$

The solution to this differential problem reads

$$y(t) = \left(y_0 - \frac{B}{r}\right) e^{r(t-t_0)} + \frac{B}{r}.$$

To verify that the above function is indeed the solution to Cauchy problem (6.3), compute its first derivative:

$$y'(t) = \left(y_0 - \frac{B}{r}\right) r\,e^{r(t-t_0)} = r\,\underbrace{\left(y_0 - \frac{B}{r}\right) e^{r(t-t_0)}}_{y(t)-\frac{B}{r}} = r\,y(t) - B,$$

which confirms that $y(t)$ satisfies the differential equation in (6.3).

Now evaluate $y(t_0)$:

$$\begin{aligned} y(t_0) &= \left(y_0 - \frac{B}{r}\right) e^{r(t_0-t_0)} + \frac{B}{r} = \left(y_0 - \frac{B}{r}\right) e^{0} + \frac{B}{r} = \\ &= y_0 - \frac{B}{r} + \frac{B}{r} = y_0, \end{aligned}$$

so $y(t)$ also satisfies the initial condition in (6.3).

When $B \neq r\,y_0$, the solution $y(t)$ is still an exponential function that, depending on r, B and y_0, can have an increasing behaviour (growing population) or decreasing behaviour (population evolving to extinction), as in the case of natural dynamics (see Fig. 6.7). In particular, since we have set $B > 0$:

1. if $r > 0$ and $B < r\,y_0$, the solution is increasing;
2. if $r > 0$ and $B > r\,y_0$, the solution is decreasing, and when $y(t)$ reaches zero the population has become extinct. Negative values of $y(t)$ (despite being mathematically legitimate) do not represent the size of the population;
3. if $r < 0$ and for any $B > 0$, the solution is decreasing and the same considerations of the previous case hold.

But whenever $B = r\,y_0$, i.e. when the quantity of harvested individuals equals the product of the growth rate and the initial quantity, the following happens:

$$y'(t) = 0 \text{ and } y(t) = \frac{B}{r} = y_0 \qquad \forall t \geq t_0.$$

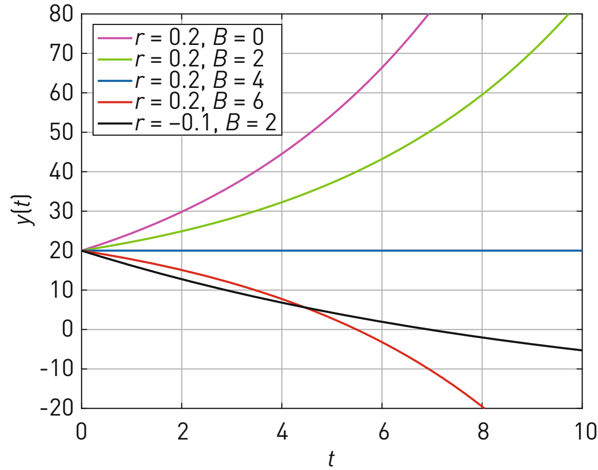


Fig. 6.7 Functions $y(t)$ solving problem (6.3) over the interval $[t_0, 10]$ with data $t_0 = 0$, $y_0 = 20$. We have a constant solution for $B = 4$ and $r = 0.2$ (in fact $B = r \cdot y_0$); increasing solutions when $r = 0.2$ and $B < 4$; decreasing solutions when $r = 0.2$ and $B > 4$ or $r = -0.1$ and $B = 2$

In this case the population stays constant in time: the natural dynamics of the population and the harvest are in *equilibrium*.

Exercises

Solve Exercise 6.2, p. 227.

6.4.3 The Interaction with the Environment: Time-Dependent Harvest (the Model)

In reality the interaction of the population with the environment is not always constant in time. It can vary according to the season or other factors, and in general it will be proportional to the size of the population (yet another improvement of the model). We may therefore model this interaction by adding the term $b(t)y(t)$ in the differential equation, thus assuming that $b(t)$ represents the *per capita variation of the population at time* t caused by external factors. Values $b(t) < 0$ will determine a decline in the population, while $b(t) > 0$ will induce a rise (as depicted in Fig. 6.8).

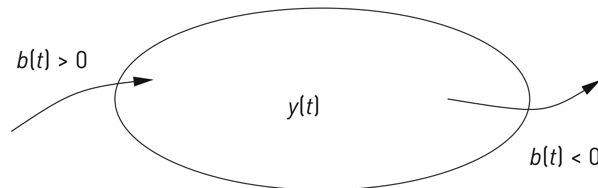


Fig. 6.8 Diagramme for the system made of one population and its interaction with the environment caused by the function $b(t)$

If b_d is a positive number and we take $b(t) = -b_d$ for every t (hence constant), the value b_d is called the *depletion rate of the population*.

The mathematical model that accounts for interactions

The mathematical model governing the dynamics of a population subject to time-dependent interactions with the environment is the differential problem

$$\begin{cases} y'(t) = r\,y(t) + b(t)y(t) = (r + b(t))y(t), & \text{if } t \geq t_0 \\ y(t_0) = y_0 \end{cases} \tag{6.4}$$

Unawareness of the solution

While Cauchy problems (6.2) and (6.3) can be solved analytically by hand, we may not be able to write the explicit solution $y(t)$ to problem (6.4). It all depends on the expression of $b(t)$. We could find ourselves in a situation similar to that of Chap. 2, Sect. 2.1 (*When we do not know the solution to the mathematical model*), and may therefore need to rely on numerical models and numerical methods.

So before proceeding any further, we should practice a bit on derivatives and on approximating Cauchy problems.

Readers not interested in the numerical methods of the following pages can jump to page 176.

6.4.4 A Numerical Method to Approximate First Derivatives

The most “problematic” object appearing in the differential equation is the first derivative of the unknown function, and we should understand how to handle it.

First derivative

We know that, by definition, the first derivative of a function f at a given point t is the limit of the difference quotient as the increment h tends to 0:

$$f'(t) = \lim_{h \to 0} \frac{f(t+h) - f(t)}{h}. \tag{6.5}$$

Unfortunately calculators are not able to compute limits: it really would be asking too much!

Therefore we have to use approximating methods to deal with the first derivative.

Consider a real-valued function f defined on a closed, bounded interval $[a, b]$

$$f : [a, b] \to \mathbb{R}.$$

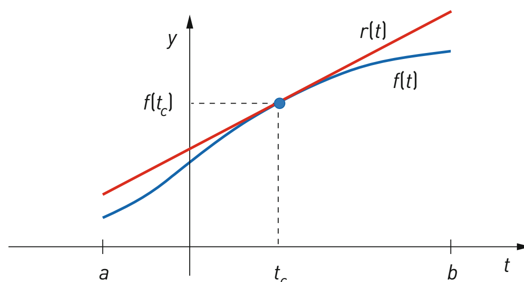


Fig. 6.9 A differentiable function at each point of the interval [a, b] and the tangent line at $t = t_c$

Suppose f is *differentiable* at every point of $[a, b]$, i.e. the first derivative (6.5) exists and is finite at every point $t \in [a, b]$. (Geometrically, this means that at each point of $[a, b]$ the function $f(t)$ admits exactly one tangent line $r(t)$ and this is not vertical, see Fig. 6.9).

Given a point $t_c \in [a, b]$, we know that the real number $f'(t_c)$ obtained from definition (6.5) is the slope of the tangent line $r(t)$ to the graph of $f(t)$ at the point of abscissa t_c. Put otherwise, the equation of the tangent line is

$$r(t) = f'(t_c)(t - t_c) + f(t_c).$$

One calls *first derivative* $f'(t)$ the function associating to every $t \in [a, b]$ the slope of the tangent line to the graph of f at t, i.e. the number $f'(t)$.

Numerical methods for approximating derivatives do not seek the expression of the first derivative as a function (as we are used to do when we apply the rules of differentiation to study a function and draw its graph). Rather, they aim for a numerical value, at a given point t_c, that approximates the exact value of the derivative $f'(t_c)$.

Forward finite difference

Let h be a sufficiently small positive number. Because of definition (6.5), we may suppose that the *difference quotient*

$$(\delta_+ f)(t_c) = \frac{f(t_c + h) - f(t_c)}{h} \tag{6.6}$$

called *forward finite difference*, is an approximation of the derivative $f'(t_c)$.

As we can see in Fig. 6.10, $(\delta_+ f)(t_c)$ is the slope of the secant line intersecting the graph of $f(t)$ at the points with abscissa t_c and $t_c + h$.

Figure 6.10 shows that the straight line with the finite difference as slope might differ from the tangent line a lot. However, as h becomes smaller, the point $t_c + h$ tends to t_c and the blue line tends to approximate the (dashed) tangent line better and better. In other terms, as h tends to zero, the finite difference (the slope of the secant line) approximates $f'(t_c)$ (the slope of the tangent line) increasingly better.

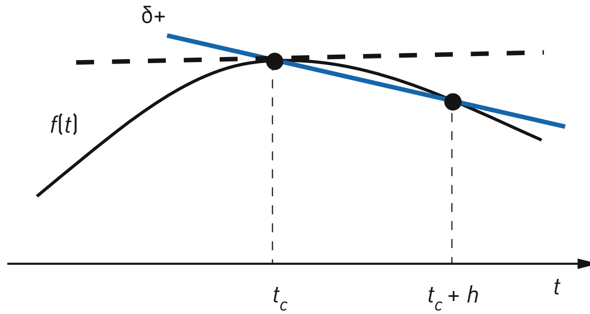


Fig. 6.10 The forward finite difference $(\delta_+ f)(t_c)$ is the slope of the line δ_+ (blue). The black dashed line is tangent to the graph of $f(t)$ at t_c, and its slope is $f'(t_c)$

We can also quantify the error made in approximating $f'(t_c)$ by the finite difference just introduced.

Error

Let the function f be twice differentiable over the interval $[a, b]$ and suppose the second derivative $f''(t)$ is continuous on $[a, b]$. Then the forward finite difference approximates the first derivative *with order one with respect to h*.

Let us recall the definition of convergent method, seen in Chap. 1. This means there exists a positive constant C only depending on f such that the numerical error behaves like h, as h tends to 0:

$$|f'(t_c) - (\delta_+ f)(t_c)| \leq Ch, \qquad \text{as} \qquad h \to 0. \tag{6.7}$$

Exercise: Computing derivatives by finite differences

Consider the function $f(t) = \sin(t)$. We know $f'\left(\frac{\pi}{6}\right) = \cos\left(\frac{\pi}{6}\right) = \frac{\sqrt{3}}{2}$:

1. determine forward finite differences approximating the value $f'(\frac{\pi}{6})$ for increasingly smaller choices of h, say $h = 10^{-1}, h = 5 \cdot 10^{-2}, h = 10^{-2}, \ldots, h = 10^{-4}$, then compute the corresponding errors;
2. plot the errors and verify that estimate (6.7) holds.

Solution. Although these easy computations can be made with a standard calculator or even by hand, we suggest you get acquainted with Octave, so to be able to solve numerically the differential equations we shall meet in the rest of the chapter.

Typing the command

```
finite_diff1
```

in the Octave window gives the values on the right in this table.

$f'(t_c)$	h	df $= (\delta_+ f)(t_c)$
0.866025	0.1	0.839604
0.866025	0.05	0.853167
0.866025	0.01	0.863511
0.866025	0.005	0.864772
0.866025	0.001	0.865775
0.866025	0.0005	0.865900
0.866025	0.0001	0.866000

Clearly $f'(t_c)$ does not change, while as h decreases the values df (the forward finite difference $(\delta_+ f)(t_c)$) in the third column tend towards the exact value $f'(t_c)$.

Our job does not stop here, since we must gauge the errors (in absolute value) made by the approximations. Otherwise said we have to understand the behaviour of $|f'(t_c) - (\delta_+ f)(t_c)|$ as h varies. Executing the command

```
finite_diff2
```

in the Octave window produces the numbers in the table below (recall that 2.642183e-02 is to be read $2.642183 \cdot 10^{-2}$).

h	ef $= \|f'(t_c) - (\delta_+ f)(t_c)\|$
0.1	$2.642183 \cdot 10^{-2}$
0.05	$1.285819 \cdot 10^{-2}$
0.01	$2.514413 \cdot 10^{-3}$
0.005	$1.253606 \cdot 10^{-3}$
0.001	$2.501443 \cdot 10^{-4}$
0.0005	$1.250361 \cdot 10^{-4}$
0.0001	$2.500144 \cdot 10^{-5}$

As h decreases, the absolute values of the errors decrease (observe the orders of magnitude first).

In particular if h decreases by one order of magnitude, also the absolute value of the error of the forward finite difference decreases by one order of magnitude.

For example, when $h = 0.01$ the error (in absolute value) is around $2.5 \cdot 10^{-3}$, and for $h = 0.001$ the error is about $2.5 \cdot 10^{-4}$.

Convergence order

We have verified that: the forward-finite-difference method converges with order 1 with respect to h and *the absolute value of the error behaves like h, as $h \to 0$*. We have verified error estimate (6.7).

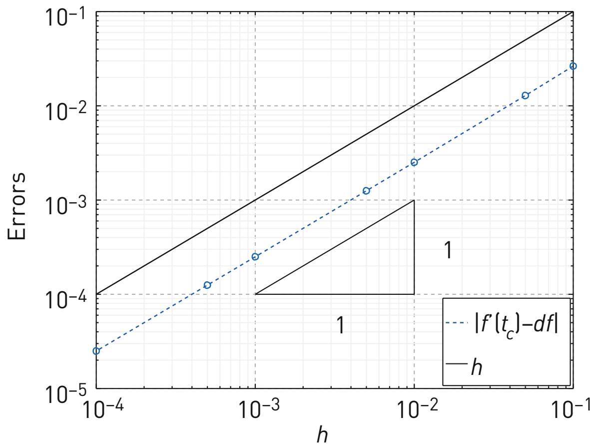


Fig. 6.11 The absolute values of the approximation errors in forward finite differences are proportional to h, as h tends to zero

Now we want to plot the result. Typing the command

```
finite_diff3
```

in the Octave window generates Fig. 6.11: in logarithmic scale, the straight line of the absolute values of the errors is parallel to the line $y(h) = h$. Hence errors are proportional to h as h tends to zero. (After the end of the problem we shall explain how to interpret logarithmic-scale plots.)

Through this exercise, apart from computing the numerical solution to our problem, we have verified the numerical method (see Chap. 1). We have, in other words, shown that the numerical solution is a good approximation of the solution of the mathematical model. □

6.4.5 How to Represent Errors

When we study a numerical method, especially its convergence (as we did in exercise *Computing derivatives by finite differences*, regarding the definition of convergent numerical method seen in Chap. 1), we might need to interpret the behaviour of numerical errors $e_n(h)$ (in absolute value) as h varies. Often these graphs are plotted using a logarithmic scale.

The Linear Scale and the Logarithmic Scale

Whilst on a *linear scale* the distance of two consecutive integers is constant, on a *logarithmic scale* what is constant is the distance between consecutive integer powers of 10.

For instance, see Fig. 6.12, on linear scale the distance between 1 and 2 is the same as the distance between 6 and 7. On logarithmic scale this is not the case: in Fig. 6.13, the distance between 10^{-1} and 1 equals the distance between 1 and 10.

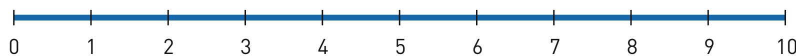


Fig. 6.12 Linear scale: the distance of consecutive integers is constant

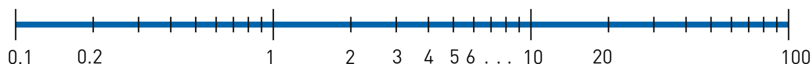


Fig. 6.13 Logarithmic scale: the distance of consecutive integer powers of 10 is constant

The advantage of the logarithmic scale over the linear one is easily explained. First, the former allows to separate very small numbers that against linear scale we would not distinguish, and this is essential when we examine quantities that are tending to zero. Secondly, the logarithmic scale enables us to easily recognise the order of convergence of a method.

Reading the Order of Convergence Off the Plot

Suppose the numerical method is convergent with some order $p \geq 1$, i.e. the numerical error it generates behaves like $\left|e_n(h)\right| \approx Ch^p$ as h tends to 0 (according to the definition of Chap. 1). Then by the properties of logarithms we may write:

$$\left|e_n(h)\right| \approx Ch^p$$
$$\underbrace{\log_{10}\left(\left|e_n(h)\right|\right)}_{y} \approx \log_{10}(Ch^p) = \underbrace{\log_{10}(C)}_{q} + p\,\underbrace{\log_{10}(h)}_{x}$$
$$y \approx r(x) = q + px.$$

We have obtained the equation of a straight line on a logarithmic scale, and the order of convergence p of a method is the slope of said line $r(x) = q + px$. Hence $p = 1$ for first-order methods, $p = 2$ for second order, and so forth.

If we wish to compare the errors generated by two methods, the error of the method with higher convergence order corresponds to the line with bigger slope.

Figure 6.14 on the left, for example, shows the lines (logarithmic scale) representing the absolute values of the errors of two numerical methods. The purpose of the little triangles is to facilitate the comparison of the slopes of the error lines: if h changes by one order of magnitude (that would be the 1 at the base of the triangle, expressing the horizontal variation of magnitude), the error in red varies by one order of magnitude (the 1 on the red vertical side gives the y-variation of magnitude). At the same time the blue error decreases by two orders of magnitude (the 2 along the vertical side of the blue triangle gives the variation of magnitude for this second error along the ordinate axis). Observe that the units on the Cartesian axes are different, so it is important to write the lengths of the sides of the triangle to avoid drawing wrong conclusions on convergence orders.

The convergence order p of a method (the slope of the line representing the error against logarithmic scale) is the ratio between the lengths of the legs (vertical to horizontal) on a logarithmic scale.

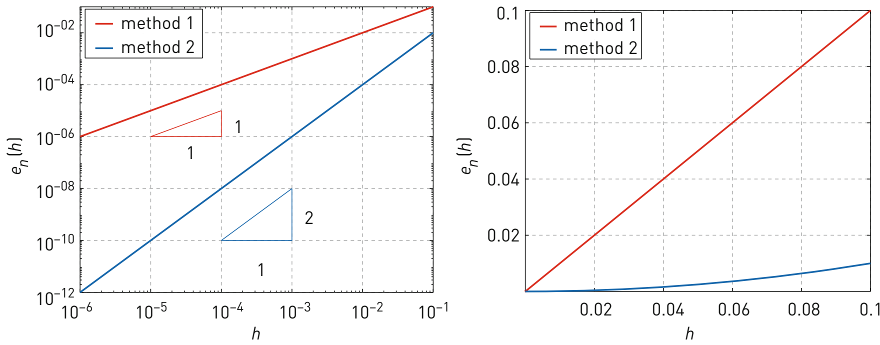


Fig. 6.14 The absolute values of the errors in the two methods: method 1 has convergence order 1, method 2 has order 2. The left picture is plotted on logarithmic scale, the right one on linear scale

So we have $p = 1$ for the red error and $p = 2$ for the blue error, and we conclude that the method whose error is given by the red line has order one, the method relative to the blue line has order two.

Picture 6.14 on the right shows the same data as the left one, but against linear scale both on the x-axis and on the y-axis. It is patent that the linear-scale representation for these data is not optimal. In fact when $x \in [10^{-6}, 10^{-2}]$ the blue curve looks very flat horizontally, despite the fact that the corresponding ordinates range over 8 orders of magnitude, from 10^{-12} to 10^{-4}.

6.4.6 The Euler Method for Approximating a Differential Equation

The goal is now to approximate the solution to Cauchy problem (6.4).

To make our analysis more general, observe that the right-hand side of the differential equation in (6.4) depends on two variables, t and $y = y(t)$ (the unknown function), and we can write it as follows:

$$f(t, y(t)) = (r + b(t))y(t).$$

Clearly $y = y(t)$ depends on t as well, so $f(t, y(t))$ is actually a function of time only. But by writing $f(t, y(t))$ we wish to stress the presence of the unknown y in the right-hand side of the differential equation.

General form of Cauchy problem

We can then solve Cauchy system (6.4) in the more general form:

$$\begin{cases} y'(t) = f(t, y(t)), & \text{if } t_0 \le t \le t_f \\ y(t_0) = y_0 \end{cases} \tag{6.8}$$

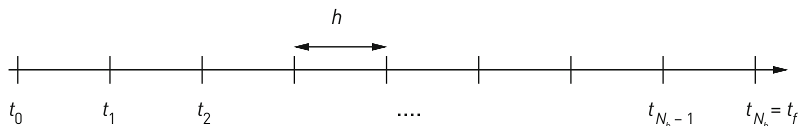


Fig. 6.15 Evenly spaced points between t_0 and t_f

Because the first derivative of a function can be approximated only point by point, also to solve problem (6.8) (involving the derivative of the unknown function) we have to work pointwise.

Our intention is not to find the expression of $y(t)$ that solves the differential equation, but to approximate its values at a set of points $t_1, t_2, \ldots, t_n, \ldots$ chosen beforehand, which will help us approximate the first derivative using finite differences, as seen earlier.

To begin with, choose a parameter $h > 0$, called the *discretisation step*, needed to approximate derivatives. Then define a collection of points between t_0 and t_f, at distance h to one another:

$$t_1 = t_0 + h, \ t_2 = t_1 + h, \ \ldots, \ t_{N_h} = t_{N_h-1} + h = t_f,$$

where the smaller the h, the larger $N_h = \dfrac{t_f - t_0}{h}$ (see Fig. 6.15).

For every n from 0 to N_h, call $y_n = y(t_n)$ the value of the exact solution (unknown) at t_n, and let u_n be a value (also unknown) that will approximate y_n. The set of numbers $\{u_0, \ u_1, \ \ldots, u_{N_h}\}$ is the *numerical solution* we seek. As y_0 is known, we clearly set $u_0 = y_0$. Figure 6.16 shows the exact solution $y(t)$ to a Cauchy problem, the exact values $y_n = y(t_n)$ and the numerical solution $\{u_0, \ u_1, \ \ldots, u_{N_h}\}$.

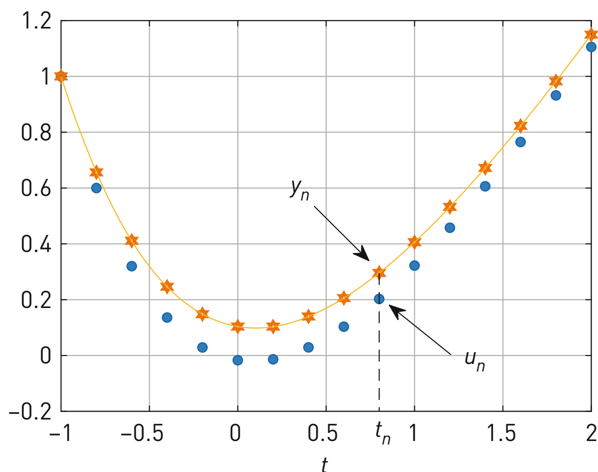


Fig. 6.16 Exact solution y_n and numerical solution u_n of a Cauchy problem

The exact solution $y(t)$ to Cauchy problem (6.8) satisfies the differential equation $y'(t) = f(t, y(t))$ for all $t \in [t_0, t_f]$, so in particular we can write

$$y'(t_n) = f(t_n, y(t_n)) \qquad \text{for every } n = 0, \ldots, N_h.$$

Start with $n = 0$: first we have

$$y'(t_0) = f(t_0, y_0),$$

then we use forward finite difference (6.6) to approximate $y'(t_0)$:

$$y'(t_0) \simeq (\delta_+ y)(t_0) = \frac{y_1 - y_0}{h},$$

and hence

$$\frac{y_1 - y_0}{h} \simeq y'(t_0) = f(t_0, y_0).$$

Now, exploiting $u_0 = y_0$, we define u_1 to be the solution of equation

$$\frac{u_1 - u_0}{h} = f(t_0, u_0),$$

or equivalently

$$u_1 = u_0 + hf(t_0, u_0).$$

Similarly, we set

$$\begin{aligned} u_2 &= u_1 + hf(t_1, u_1) \\ u_3 &= u_2 + hf(t_2, u_2) \\ \ldots \end{aligned}$$

Euler method

In this way we obtain the general form of *the Euler method*:

$$\begin{aligned} u_0 &= y_0 \\ u_{n+1} &= u_n + hf(t_n, u_n) \qquad \text{for } n = 0, \ldots, N_h - 1. \end{aligned} \tag{6.9}$$

The Euler method is therefore a numerical method to approximate the solution of a differential equation.

The following question becomes natural at this point: What is the best value of N_h for computing the numerical solution?
Equivalently: how does one choose the discretisation step h?

The answer is not clear-cut. It depends on which accuracy we require in order to approximate the exact solution to our problem, i.e. which numerical error $e_n(h)$ we are willing to accept, as we set out to explain.

Exercise: Computing the numerical solution to the Cauchy system

We want to approximate the solution to the Cauchy problem

$$\begin{cases} y'(t) = \dfrac{\cos(t) - y(t)}{t+1}, & \text{if } 0 \leq t \leq 10 \\ y(0) = 0 \end{cases}$$

using Euler method (6.9) and three different values for N_h:
$N_h = 10$, $N_h = 100$ and $N_h = 1000$.

Solution. We have translated method (6.9) into the function `euler` (see Function 6.1, page 194). We have also prepared a second function, `cauchy_problem` (see Function 6.2, page 195), in which:

- we specify the data of the Cauchy problem to be solved: the initial time $t_0 = 0$, the final time $t_f = 10$, the initial datum $y_0 = 0$ and the function $f(t, y) = \dfrac{\cos(t) - y}{t+1}$ (these come from comparing this specific problem with the general form (6.8));
- we solve the Cauchy problem numerically by calling the `euler` function, with N_h chosen in due course;
- we plot the numerical solution.

Let us begin by computing the numerical solution with $N_h = 10$ time steps, corresponding to discretisation step $h = \dfrac{t_f - t_0}{N_h} = 1$. Typing the Octave command

```
close all
cauchy_problem(10)
```

gives the numerical solution of Fig. 6.17, left: the blue dots are the values $u_0, u_1, \ldots, u_{10}$ corresponding to times $t_0, t_1, \ldots, t_{10}$.

Then we compute the numerical solution with $N_h = 100$ (corresponding to discretisation step $h = \dfrac{t_f - t_0}{N_h} = 0.1$) by typing the command

```
cauchy_problem(100)
```

and finally with $N_h = 1000$ (so $h = \dfrac{t_f - t_0}{N_h} = 0.01$) using

```
cauchy_problem(1000)
```

□

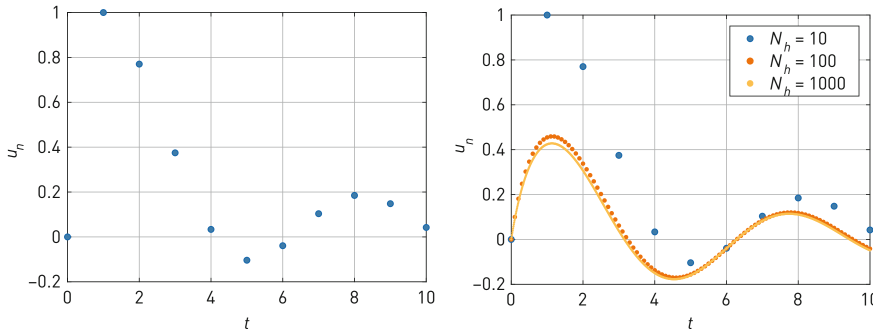


Fig. 6.17 The numerical solution of the Cauchy problem of exercise *Computing the numerical solution of a Cauchy problem*, computed by the Euler method and $N_h = 10$ (step $h = 1$) (left), $N_h = 10$, $N_h = 100$ and $N_h = 1000$ (right)

The numerical solutions obtained with the three choices of N_h are plotted in Fig. 6.17, right. They are manifestly different, so we ask: which one is the most accurate?

6.4.7 Error of the Euler Method

Convergence order of the Euler method

The Euler method is based on the approximation of the first derivative by finite difference (6.6), which is accurate of order one with respect to h, as $h \to 0$.

Hence the numerical solution found by the Euler method is also accurate of order one in h, as $h \to 0$. Put otherwise, there exists a constant $C > 0$ independent of h such that

$$|y_n - u_n| \leq Ch \qquad \text{for every } n = 0, \ldots, N_h. \tag{6.10}$$

We say that *the Euler method converges with order 1 with respect to h.*

At any time t_n the error between the numerical solution u_n and the exact solution y_n tends to zero like h, as $h \to 0$. Hence the smaller h is, the more accurate the numerical solution u_n turns out to be. Consequently the most precise result among those found in exercise *Computing the numerical solution to the Cauchy problem* is the one obtained with the smallest h, i.e. $h = \dfrac{1}{1000}$ (corresponding to the largest N_h, i.e. $N_h = 1000$).

By further increasing N_h, the numerical solution will become more accurate, but beware: as N_h gets bigger, our program needs to execute more and more operations, and hence we will have to wait longer for an answer.

The Euler method is a very powerful and useful tool that suggests how to compute the numerical solution to any Cauchy problem of the form (6.4).

Unless we want to solve it by hand, the Euler method must be implemented in a computer program, and the `euler` function (Function 6.1 on page 194) is one example.

In case we know the exact solution to the Cauchy problem, e.g. problems (6.2), page 160 and (6.3), page 163, besides computing the numerical solution we can also measure the errors produced by the numerical method.

Exercises

Solve Exercise 6.3, p. 228.

6.4.8 The Interaction With the Environment: Time-Dependent Harvest (the Numerical Solution)

At this point we have all the necessary instruments to compute an approximated solution to model (6.4), page 165, for whichever function $b(t)$ expressing the interaction with the environment. To understand how to compute the numerical solution of a Cauchy problem let us consider the following situation.

Exercise: Aeolian dolphins (2): the effect of the environment

Consider once again the dolphin population in the Aeolian waters (see exercise *Aeolian dolphins*, page 161), which in July 2017 was of 100 individuals. Suppose that number now decreases due to human intervention. Some dolphins are caught in fishing nets or killed by the blades of speedboat propellers according to a yearly trend: during the summer (July-August) there are more deaths, while in winter (December-January) the losses are fewer. The function $b(t)$ describing the reduction of the number of dolphins caused by the interaction with the environment is[6]

$$b(t) = -0.03(2 - e^{-5(\sin(t\pi))^2}),$$

with t measured in years.
One would like to know how the dolphin population evolves in the period going from July 2017 to July 2025 in this new context.

Solution. Let us start by plotting the function $b(t)$ (see Fig. 6.18; to plot the graph of $b(t)$ in Octave see Sect. 3.1, Chap. 3). The function $b(t)$ is always negative, and as noted it is periodic with period one year. It models the percentage of dolphins, over time, that is subtracted due to human actions.

The mathematical model is Cauchy problem (6.4) together with the data specified in the text, that is

$$\begin{cases} y'(t) = 0.06\, y(t) - 0.03(2 - e^{-5(\sin(t\pi))^2})y(t), & \text{if } 2017.5 \le t \le 2025.5 \\ y(2017.5) = 100. \end{cases}$$

[6]It can be shown that the exponent $-5(\sin(t\pi))^2$ is periodic, with period 1. Hence the function $b(t)$ is periodic with period 1, that is: $b(t+1) = b(t)$ for all $t \ge 0$.

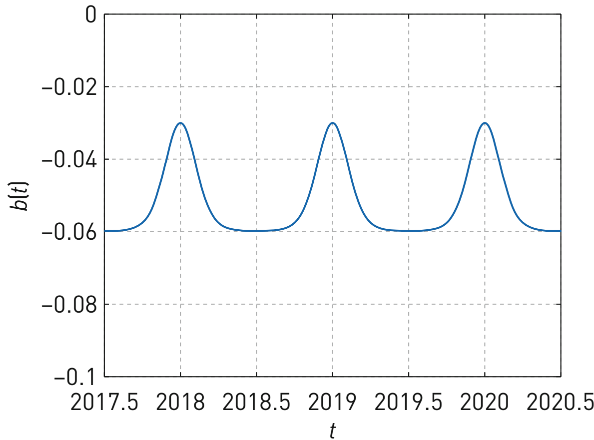


Fig. 6.18 The function $b(t)$ defined in exercise *Aeolian dolphins (2): the effect of the environment*. It is periodic with period equal 1 year

- The *data* are $r = 0.06$, $t_0 = 2017.5$, $y_0 = 100$ and $b(t) = -0.03(2 - e^{-5(\sin(t\pi))^2})$;
- the *mathematical model* is Cauchy problem (6.4), page 165;
- the *solution* is the value of y at time $t = 2025.5$.

Notice we have replaced $t \geq t_0$ by the condition that t belong in the interval between $t_0 = 2017.5$ (the instant at which we know the initial condition) and $t_f = 2025.5$ (the final instant at which we are seeking the solution).

Unfortunately there are no formulas for calculating the exact solution of this model (as of many others), so we have to use numerical methods. To compute the numerical solution, fix an integer $N_h > 0$ representing the number of subintervals partitioning $[t_0, t_f]$, to which the time variable t belongs. They all have the same length $h = \dfrac{t_f - t_0}{N_h}$. The larger N_h we take (the smaller h), the more complex will it be to compute the numerical solution, and at the same time the more precise the solution will become.

To solve the given Cauchy problem numerically let us call the `dolphins2` function (see Function 6.3, page 195). After defining the data of the problem, within it we call the `euler` function to compute and plot the numerical solution: first we take $N_h = 16$ time steps, corresponding to a discretisation step $h = \dfrac{t_f - t_0}{N_h} = \dfrac{1}{2}$ year, so six months. Then we take $N_h = 96$ time steps, so $h = \dfrac{1}{12}$ of a year, i.e. one month. Typing the command

```
close all
unfinal1=dolphins2(16)
```

gives

```
unfinal1 = 112.74
```

representing the approximation of the required solution, i.e. the number of dolphins in July 2025, computed with 16 time steps. Instead, by typing

```
unfinal2=dolphins2(96)
```

we obtain

```
unfinal2 = 106.69
```

which is the approximation of the solution (the number of dolphins in July 2025) computed with 96 time steps.

Figure 6.19 shows the numerical solutions found with the two choices of N_h: in blue the solution with $N_h = 16$ $\left(\text{so } h = \dfrac{1}{2}\right)$, in red the one with $N_h = 96$ $\left(\text{so } h = \dfrac{1}{12}\right)$. They are quite different.

Recall that by error estimate (6.10), the smaller h is, the more accurate the approximation u_n becomes. Therefore the more precise result (of the two that were computed) is the one with smaller h, so $h = \dfrac{1}{12}$ (corresponding to larger N_h, i.e. $N_h = 96$).

Now, if we wanted to compute another numerical solution with $N_h = 192$ $\left(\text{corresponding to } h = \dfrac{1}{24} \text{ of a year, roughly 15 days}\right)$, we should execute `dolphins2` again:

```
unfinal3=dolphins2(192)
```

and obtain 106.69 once more.

Further increasing N_h will make the decimal digits of the numerical solution more accurate. However in this particular problem we might content ourselves with approximating properly only the integer part. The numerical solution to problem

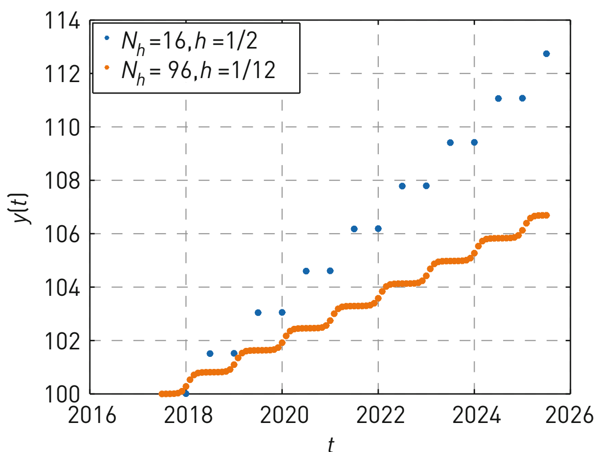


Fig. 6.19 The numerical solutions of exercise *Aeolian dolphins (2): the effect of the environment* computed by the Euler method and $N_h = 16$ $\left(\text{step } h = \frac{1}{2}\right)$ and $N_h = 96$ $\left(\text{step } h = \frac{1}{12}\right)$

Aeolian dolphins (2): the effect of the environment is the roundoff of the value found by `dolphins2` with $N_h = 96$.

In conclusion, according to our model the number of dolphins in July 2025 will be 107 (after rounding decimals in 106.69). □

Exercises

Solve Exercise 6.4, p. 229 and Exercise 6.5, p.231.

6.4.9 An Even More Realistic Model: The Competition Within the Population

Let us begin with mathematical model (6.4) on page 165, which describes the dynamics of a population subject to interactions with the surrounding environment (like fishing). We enhance it by supposing that the population growth is affected by limited environmental resources, plus other factors that are specific to the population itself.

We would like to model, for instance, the fact that as the population grows its members will start competing for the limited food and vital space available, so eventually the internal antagonism will stymie a large population $y(t)$.

This means that the constant growth rate r, which we assume positive, should be replaced by a rate depending on the size of the population. Precisely, we should replace it by the variable growth

$$r - \frac{r}{K} y(t).$$

The constant $K > 0$ is the *carrying capacity of the environment*, i.e. the largest population that the environment can sustain. An infinite value of K means unlimited resources. In such a case, in fact, the term $r\,\frac{y(t)}{K}$ is infinitely small and hence insignificant compared to the natural growth rate r

$$r - \frac{r}{K} y(t) \simeq r.$$

If, instead, the resources are limited (say, there is little food or the space is inadequate for that number of individuals), then the smaller the K the more important $\frac{r\,y(t)}{K}$ becomes compared to the natural growth rate r, and the more internal competition for survival the population will face.

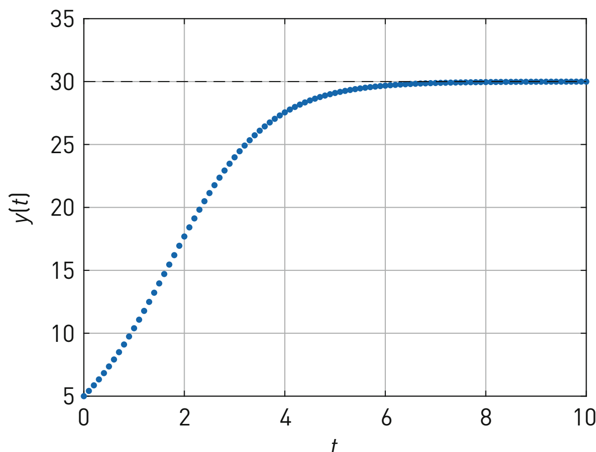


Fig. 6.20 The solution to model (6.11) corresponding to growth rate $r = 0.1$. The carrying capacity is $K = 30$ and we fixed $y(0) = y_0 = 5$

The mathematical model based on the logistic equation

The model describing the dynamics of a population with limited resources and no environmental interactions is[7]

$$\begin{cases} y'(t) = r\, y(t) - \dfrac{r}{K} y^2(t), & \text{if } t \geq t_0, \\ y(t_0) = y_0. \end{cases} \tag{6.11}$$

As time goes by the solution tends asymptotically to the carrying capacity K, as we can see in Fig. 6.20.

Let us go back and include the interaction with the environment as well.

The mathematical model that accounts for competition and harvest

The mathematical model describing the evolution of a population with growth rate $r > 0$, including both the internal competition for limited resources as well as the interaction with the environment, reads:

$$\begin{cases} y'(t) = r\, y(t) - \dfrac{r}{K} y^2(t) + b(t) y(t), & \text{if } t \geq t_0, \\ y(t_0) = y_0. \end{cases} \tag{6.12}$$

As for Cauchy problem (6.4), page 165, whether we are able to use a method yielding the explicit exact solution will depend on the particular function $b(t)$.

[7] This model is known as *the logistic equation*, or *Verhulst equation* after the Belgian mathematician and statistician Pierre François Verhulst (1804–1849).

But now this does not scare us anymore: we have a numerical method (the Euler method) allowing us to approximate the exact solution of our problem with the accuracy we want (by choosing the discretisation parameter h suitably or, equivalently, the number of time steps N_h).

Exercise: Aeolian dolphins (3): the influence of the environment and competition

Consider once again the dolphin population in the Aeolian waters (see exercise *Aeolian dolphins (2): the effect of the environment*), which in July 2017 consisted of 100 individuals. Suppose there still is interaction with the environment, dictated by the function $b(t) = -0.03(2 - e^{-5(\sin(t\pi))^2})$. Further assume that *the carrying capacity of the population* is $K = 150$. We wish to study how the number of dolphins evolves between July 2017 and July 2025 in this new context.

Solution. In this particular case the Cauchy problem of the model (6.12) on page 180, reads:

$$\begin{cases} y'(t) = 0.06y(t) - \dfrac{0.06}{150}y^2(t) - 0.03(2 - e^{-5(\sin(t\pi))^2})y(t) \\ \qquad\qquad\qquad\qquad \text{if } 2017.5 \le t \le 2025.5 \\ y(2017.5) = 100. \end{cases}$$

- The *data* are $r = 0.06$, $t_0 = 2017.5$, $y_0 = 100$, $K = 150$ and $b(t) = -0.03(2 - e^{-5(\sin(t\pi))^2})$;
- the *mathematical model* is Cauchy problem (6.12), page 180;
- the *solution* is the value of y at time $t = 2025.5$.

Since there are no formulas to find the solution $y(t)$ to the Cauchy problem, we will compute the numerical solution with the Euler method, using $N_h = 96$ time steps $\left(\text{corresponding to a discretisation time } h = \dfrac{1}{12}, \text{ i.e. one month}\right)$. To this end we have prepared the `dolphins3` function (see Function 6.4, page 196). The Octave command:

```
close all
unfinal=dolphins3(96)
```

produces:

```
unfinal =  80.15
```

and the numerical solution over the whole time interval is plotted in Fig. 6.21.

We might summarise as follows: given the limited resources (expressed by the carrying capacity $K = 150$) and the environmental interaction (modelled by $b(t)$), in July 2025 there will be 80 dolphins left. This is the numerical solution computed by the Euler method with $N_h = 96$. We deduce from the plot that in absence of changes in the system the population will eventually become extinct. □

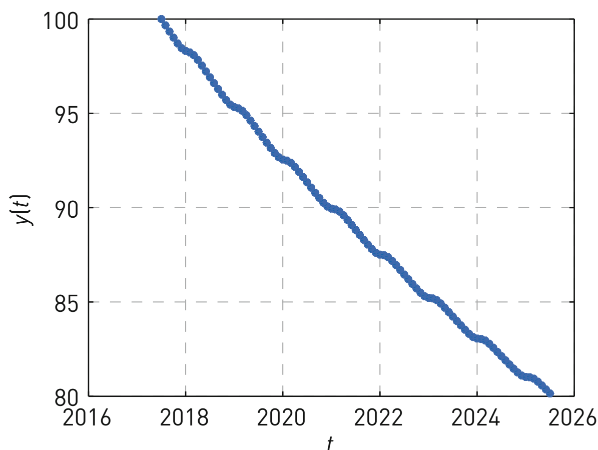


Fig. 6.21 The numerical solution to the Cauchy problem of exercise *Aeolian dolphins (3): the influence of the environment and competition*, computed by the Euler method with $N_h = 96$

At this point we should be satisfied with the mathematical model we have built to study a single population. Now we are ready to truly up the game: we shall abandon the (restrictive) hypothesis of having just one population and pass to consider the interaction of two populations.

6.5 The Dynamics of Two Populations

To model the dynamics of two interacting populations we shall adopt simplifications 1–5 discussed on page 156, and make use of everything we have learnt so far regarding the evolution of a single species.

Let us start from the assumption that the size $p(t)$ of the prey population evolves under the most complete model yet, namely system (6.12) on page 180. This is the model accounting for both the internal competition and the interaction with the environment.

As for the size $s(t)$ of the predator population, we shall suppose it evolves under a model similar to (6.12), in which the growth rate r is negative and $-\dfrac{r}{K}y^2(t)$ is replaced by an analogous term $-\sigma y^2(t)$ (with $\sigma > 0$ being a suitable parameter). The latter represents another manifestation of the internal competition caused by the limited resources.

In the following pages we will first of all try to understand how to model the interaction of the two species by imagining they constitute an isolated system. Then, from page 188, we will include fishing, which will give us a more fitting interaction pattern between the two populations and the environment.

6.5.1 The Prey-Predator System (the Model)

Suppose that the prey fish live only on plankton, and predators only on prey fish. As we are examining an isolated prey-predator system, the only interaction of preys with the environment is actually the one with the predator population, and conversely, predators only interact with preys (as sketched in Fig. 6.22).

To model these interactions we declare that among all prey-predator encounters only a certain fraction is favourable to predators (meaning a prey is caught, and eaten). We write α, a real number between 0 and 1, for the *per-capita predation rate*, i.e. the percentage of prey-predator encounters that end well for a predator (and not so well for its prey) in the unit of time.

To describe how the prey population evolves we take model (6.12): we assume preys grow by a rate $r_p > 0$, that their resources are limited by the carrying capacity K_p, and that they interact with the other species (which to them represents the 'environment', in the isolated system) as prescribed by a function $b_p(t)$.

The interaction with predators represents a loss for the prey population, so b_p will be negative (the sizes s, p of the populations cannot be negative):

$$b_p(t) = -\alpha s(t).$$

Therefore the function describing the reduction of preys due to predators is proportional to the number of predators s by a factor equal to the per-capita predation rate α.

Having set, for simplicity, $\sigma_p = \dfrac{r_p}{K_p}$, the differential equation governing the evolution of preys comes from Cauchy problem (6.12) with $p(t)$ replacing $y(t)$, r_p replacing r, σ_p instead of $\dfrac{r}{K}$ and $b_p(t) = -\alpha s(t)$ in place of $b(t)$:

$$p'(t) = r_p p(t) - \sigma_p p^2(t) - \alpha s(t) p(t).$$

As for predators, they have no source of food other than the outcome of the interaction with preys, so under natural dynamics we have growth factor $r_s < 0$. We further suppose there is internal competition due to food scarcity (preys), described in the model by $-\sigma_s s^2(t)$ (where $\sigma_s > 0$ is the internal competition coefficient of predators), and also a certain interaction with preys, accounted for by the term $b_s(t)s(t)$.

Since prey-predator encounters are beneficial to the latter species, the function $b_s(t)$ is positive. Next, let us quantify the favourable effect on predators of successful

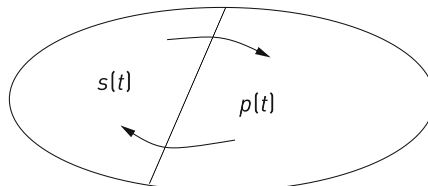


Fig. 6.22 Diagramme of the system of two interacting populations isolated from the surrounding environment

hunting by a positive coefficient μ, a real number between 0 and 1 called the *prey-to-predator conversion rate*. All-in-all

$$b_s(t) = \mu\,\alpha\,p(t),$$

whence the function describing the rise of predators is proportional to the number of preys p by a factor given by the product of the predation rate α and the conversion rate μ.

The evolution of predators is therefore modelled by the differential equation

$$s'(t) = r_s s(t) - \sigma_s s^2(t) + \mu\,\alpha\,p(t)s(t).$$

This equation comes from Cauchy problem (6.12), where we have replaced $y(t)$ by $s(t)$, r by r_s, $\dfrac{r}{K}$ by σ_s and $b(t)$ by $b_s(t) = \mu\alpha p(t)$.

The mathematical model for an isolated prey-predator system

The mathematical model describing the isolated prey-predator system is[8]

$$\begin{cases} p'(t) = \big(r_p - \sigma_p p(t) - \alpha\,s(t)\big)\;p(t), & \text{if } t_0 \le t \le t_f \\ s'(t) = (r_s - \sigma_s s(t) + \mu\,\alpha\,p(t))\;s(t), & \text{if } t_0 \le t \le t_f \\ p(t_0) = p_0 \\ s(t_0) = s_0. \end{cases} \tag{6.13}$$

To find the numerical solution of system (6.13) we will use the Euler method of page 171, and adapt it to suit a system of two differential equations.

Below we shall explain how to proceed. Readers not interested in the algorithmic part may jump directly to page 185.

6.5.2 Numerical Approximation of a System of Differential Equations

Just as we did on page 171 for one differential equation, we choose the discretisation step $h > 0$, find the number of time steps $N_h = \dfrac{t_f - t_0}{h}$ and define evenly spaced points $t_0, t_1, \ldots, t_{N_h}$.

Then we compute the values $\{u_0, u_1, \ldots, u_{N_h}\}$ (u_n approximates $p_n = p(t_n)$ for every $n = 0, \ldots, N_h$) and $\{v_0, v_1, \ldots, v_{N_h}\}$ (v_n approximates $s_n = s(t_n)$ for every $n = 0, \ldots, N_h$) by substituting the derivatives $p'(t_n)$ and $s'(t_n)$ with the corresponding forward-finite-difference formulas

$$p'(t_n) \simeq \frac{p_{n+1} - p_n}{h}, \qquad s'(t_n) \simeq \frac{s_{n+1} - s_n}{h}.$$

[8] It is a system of two differential equations. The case with $\sigma_s = \sigma_p = 0$ is known as the *Lotka-Volterra prey-predator model.*

Given p_0 and s_0, at any time t_n, $n = 0, \ldots, N_h - 1$, we have

$$\begin{cases} \dfrac{p_{n+1} - p_n}{h} \simeq p'(t_n) = (r_p - \sigma_p p_n - \alpha s_n) p_n \\ \dfrac{s_{n+1} - s_n}{h} \simeq s'(t_n) = (r_s - \sigma_s s_n + \mu\alpha p_n) s_n. \end{cases}$$

Replacing p_n with u_n and s_n with v_n, then setting $u_0 = p_0$, $v_0 = s_0$, gives

$$\begin{cases} \dfrac{u_{n+1} - u_n}{h} = (r_p - \sigma_p u_n - \alpha v_n) u_n & \text{for } n = 0, \ldots, N_h - 1 \\ \dfrac{v_{n+1} - v_n}{h} = (r_s - \sigma_s v_n + \mu\alpha u_n) v_n & \text{for } n = 0, \ldots, N_h - 1. \end{cases}$$

Equivalently:

$$\begin{cases} u_0 = p_0,\ v_0 = s_0 \\ u_{n+1} = u_n + h \cdot (r_p - \sigma_p u_n - \alpha v_n) u_n & \text{for } n = 0, \ldots, N_h - 1 \\ v_{n+1} = v_n + h \cdot (r_s - \sigma_s v_n + \mu\alpha u_n) v_n & \text{for } n = 0, \ldots, N_h - 1. \end{cases}$$

The numerical solutions $\{u_0, \ldots, u_{N_h}\}$ and $\{v_0, \ldots, v_{N_h}\}$ of this system can be computed by calling the `euler2` function (see Function 6.5, page 196). As a matter of fact this function was actually written to solve an arbitrary system of two equations

$$\begin{cases} p'(t) = f_1(t, p(t), s(t)),\ t_0 \le t \le t_f \\ s'(t) = f_2(t, p(t), s(t)),\ t_0 \le t \le t_f \\ p(t_0) = p_0 \\ s(t_0) = s_0 \end{cases}$$

and not just system (6.13). The latter is a special case of the above general situation, in which:

$$f_1(t, p(t), s(t)) = (r_p - \sigma_p p(t) - \alpha s(t)) p(t)$$

$$f_2(t, p(t), s(t)) = (r_s - \sigma_s s(t) + \mu\alpha p(t)) s(t).$$

6.5.3 The Prey-Predator System (the Numerical Solution)

At this point we possess the tools to compute an approximate solution to model (6.13), page 184.

Exercise: Hakes and sharks

Consider an isolated system of hakes (preys) and sharks (predators) characterised by these parameters:

- growth rates: $r_p = 0.6$ and $r_s = -0.2$;
- prey carrying capacity: $K_p = 3 \cdot 10^5$;

- internal competition coefficient of predators: $\sigma_s = 10^{-4}$;
- per-capita predation rate: $\alpha = 0.001$;
- conversion rate: $\mu = 0.01$.

We want to study the evolution of the two populations over an 80-year span, with a starting situation of 5000 hakes and 40 sharks.

Solution. The mathematical model regulating problem *Hakes and sharks* is Cauchy problem (6.13) together with the specified data:

$$\begin{cases} p'(t) = (0.6 - 2 \cdot 10^{-6} p(t) - 10^{-3} s(t))\, p(t), & \text{if } 0 \le t \le 80 \\ s'(t) = (-0.2 - 10^{-4} s(t) + 10^{-5} p(t))\, s(t), & \text{if } 0 \le t \le 80 \\ p(0) = 5000 \\ s(0) = 40. \end{cases}$$

- The *data* are $r_p = 0.6$, $r_s = -0.2$, $K_p = 3 \cdot 10^5$, $\sigma_p = \dfrac{r_p}{K_p} = 2 \cdot 10^{-6}$, $\sigma_s = 10^{-4}$, $\alpha = 10^{-3}$, $\mu = 10^{-2}$; initial time $t_0 = 0$, final time $t_f = 80$ (unit = one year), initial populations $p_0 = 5000$, $s_0 = 40$;
- the *mathematical model* is Cauchy problem (6.13), page 184;
- the *solution* consists of the functions $p(t)$, $s(t)$ over the time interval [0, 80].

We have prepared the `hakes_sharks` function (Function 6.6, page 197) to find the numerical solution of this Cauchy problem by the Euler method, in which:

- we specify the data of problem (6.13): the initial time $t_0 = 0$ and final time $t_f = 80$, the initial data $p_0 = 5000$, $s_0 = 40$, the various coefficients and the functions $f_1(t,p,s)$, $f_2(t,p,s)$ (see page 185);
- we solve numerically the Cauchy problem by the `euler2` function with N_h given in input (recall that the larger N_h, the more accurate the numerical solution, but the greater number of operations the computer needs to do);
- we plot the numerical solution.

Fix $N_h = 1000$ (this is large enough to guarantee a good accuracy). Typing in Octave the command

```
hakes_sharks(1000)
```

gives the numerical solution of Fig. 6.23.

Note that the sizes of the two species vary quite a lot: initially, preys grow fast in number and predators slightly decrease, but as soon as preys become very abundant (at around $t = 10$), they turn into a large food resource for predators, which therefore increase considerably. At this point the high number of sharks determines a drop in hake numbers. But as soon as food dwindles, sharks become more scarce. The system therefore follows a cyclic pattern of expansion followed by contraction. The oscillations tend to wither in time, until the system reaches a stable equilibrium, at roughly 26000 hakes and 570 sharks.

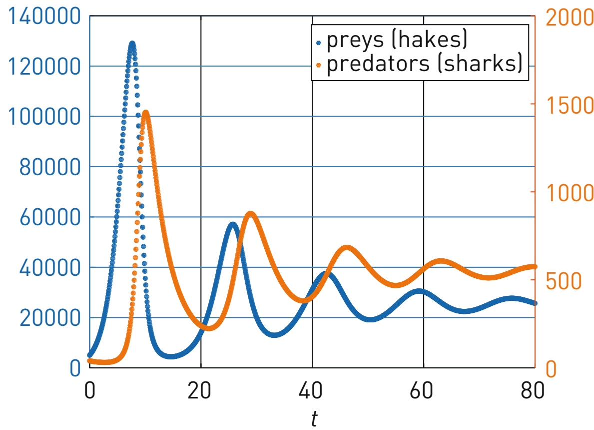


Fig. 6.23 The numerical solution to exercise *Hakes and sharks*, i.e. the size of the populations (preys and predators). The left scale (blue) refers to preys, the right one (orange) is for predators

Towards the equilibrium

We conclude that the number of predators evolves in function of the number of preys, and conversely.

As time goes by, the system tends to a stable equilibrium in which the number of hakes and sharks stays practically constant.

The second picture produced by the `hakes_sharks` function (Fig. 6.24) shows the *evolution of the number of predators in terms of the number of preys*. The plot,

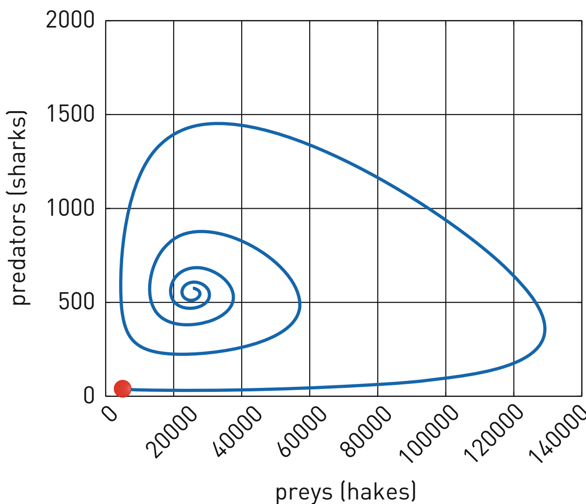


Fig. 6.24 The trajectory $s(p)$ of problem *Hakes and sharks*. The red dot represents the initial condition of the prey-predator system (6.14), i.e. the point $(p_0, s_0) = (5000, 40)$

in other words, displays the curve $s(p)$, called the *trajectory of the prey-predator system*. It behaves like a convergent spiral, starting at the point $(p_0, s_0) = (5000, 40)$ at time $t = 0$ (the initial condition of the Cauchy problem, marked by the red dot), and ending at $(25648, 575)$. The endpoint encodes the sizes s and p of the populations at the final time $t_f = 80$. □

6.5.4 An Even More Realistic Model: The Prey-Predator System with Harvest

Now we pass to the final mathematical model. Beside the mutual interaction between populations, it will account for the interaction with the environment as well, for instance fishing (Fig. 6.25).

Based on the experience we have acquired with one species subject to variable harvest (see page 164), we know that if $y(t)$ represents the size of the population at time t, the fishing activity (i.e. the draw of individuals from the population) may be described by a term like $b(t)y(t)$, where $b(t)$ is a negative function, in general depending on time.

We assume fishing does not discriminate between the two populations. That means we have to use the same function $b(t)$ for both preys and predators.

So now we modify the prey-predator model (6.13), page 184, by adding the terms $b(t)p(t)$ in the first equation and $b(t)s(t)$ in the second.

The mathematical model for a prey-predator system with harvest

The mathematical model for the prey-predator system with harvest reads

$$\begin{cases} p'(t) = \big(r_p - \sigma_p p(t) - \alpha s(t) + b(t)\big)\, p(t) & \text{if } t_0 \le t \le t_f \\ s'(t) = (r_s - \sigma_s s(t) + \mu\alpha p(t) + b(t))\, s(t) & \text{if } t_0 \le t \le t_f \\ p(t_0) = p_0 \\ s(t_0) = s_0. \end{cases} \tag{6.14}$$

In case $b(t)$ is constant (and negative) we set $b(t) = -b_d$, where $b_d > 0$ is the depletion rate.

Let us apply the new model to the hake-shark system.

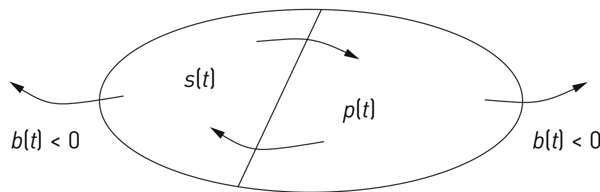


Fig. 6.25 A diagramme of the system of two interacting populations ($s(t)$ and $p(t)$) in presence of fishing (modelled by the function $b(t)$)

Exercise: Hakes and sharks (2): fishing

Consider the prey-predator system of hakes and sharks of exercise *Hakes and sharks*. We wish to study how their numbers evolve over 80 years, starting from an initial situation of 5000 hakes and 40 sharks, and assuming there is constant fishing with the same depletion rate for the two populations, so $b(t) = -b_d$. At first we will take $b_d = 0.3$, then $b_d = 0.1$.

Solution. The mathematical model for exercise *Hakes and sharks (2): fishing* is Cauchy problem (6.14). We rewrite the latter and insert the numerical data except for $b(t) = -b_d$, which will be a parameter. This will allow us to study several cases simultaneously:

$$\begin{cases} p'(t) = (0.6 - 2 \cdot 10^{-6} p(t) - 10^{-3} s(t) - b_d)\, p(t) & 0 \le t \le 80 \\ s'(t) = (-0.2 - 10^{-4} s(t) + 10^{-5} p(t) - b_d)\, s(t) & 0 \le t \le 80 \\ p(t_0) = 5000 \\ s(t_0) = 40. \end{cases}$$

- The *data* are $r_p = 0.6$, $r_s = -0.2$, $K_p = 3 \cdot 10^5$, $\sigma_p = \dfrac{r_p}{K_p} = 2 \cdot 10^{-6}$, $\sigma_s = 10^{-4}$, $\alpha = 10^{-3}$, $\mu = 10^{-2}$, and b_d corresponding to the case considered; initial time $t_0 = 0$, final time $t_f = 80$ (unit = one year), initial population sizes $p_0 = 5000$, $s_0 = 40$;
- the *mathematical model* is the Cauchy problem (6.14) of page 188;
- the *solution* is given by the functions $p(t)$ and $s(t)$ over the interval $t \in [0, 80]$.

We have prepared a new function, `hakes_sharks_fishing` (see Function 6.7, page 198) containing the data of the Cauchy problem, and we have chosen to look at three cases, corresponding to the values $b_d = 0.1$, $b_d = 0.3$ and $b_d = 0$ (so to compare with exercise *Hakes and sharks* without harvest). Then we have called the `euler2` function. A larger b_d corresponds to greater harvest over the same time interval.

Fix $N_h = 1000$ (this is large enough to warrant a good accuracy for the solution). Typing in Octave the command

```
hakes_sharks_fishing(1000)
```

returns

```
bd=0.3,  52950 hakes and    200 sharks at time t=80
bd=0.1,  33971 hakes and    450 sharks at time t=80
bd=0.0,  25648 hakes and    575 sharks at time t=80
```

and generates the numerical solutions of Fig. 6.26. The left picture shows the number of prey in time, while the right picture shows predators. In both plots we have three curves, one for each value of b_d.

When fishing is eased (passing from $b_d = 0.3$ to $b_d = 0$) *prey numbers tend to contract*. The blue curve (more fishing) for $t \ge 30$ lies above the orange curve (less fishing), which in turn lies above the yellow curve (no fishing at all).

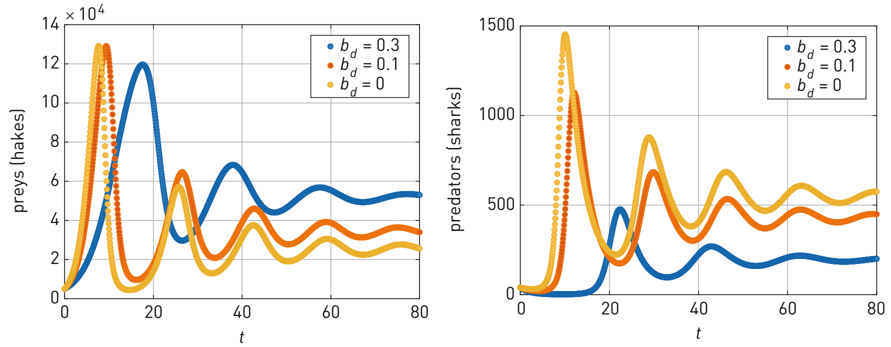


Fig. 6.26 Size evolution of prey (left) and predators (right) in exercise *Hakes and sharks (2): fishing* as time goes by, for two depletion rates (indicating a fishing activity). The results for $b_d = 0.1$ are in orange, for $b_d = 0.3$ in blue. In yellow the solutions to exercise *Hakes and sharks* where no fishing is present, for comparison

Table 6.1 The numerical results of exercises *Hakes and sharks* and *Hakes and sharks (2): fishing*

b_d	$p(t_f)$	$s(t_f)$	$p(t_f) + s(t_f)$	$R(t_f)$
0.3	52950	200	53150	$0.0038 \simeq 0.4\%$
0.1	33971	450	34421	$0.0131 \simeq 1.3\%$
0.0	25648	575	26224	$0.0219 \simeq 2.2\%$

But *a cut-back in fishing* (from $b_d = 0.3$ to $b_d = 0$) also causes *predators to increase*, in contrast to preys: for $t \geq 30$ the blue curve (more fishing) lies below the orange curve (less fishing), which is below the yellow curve (no fishing).

Table 6.1 summarises the numbers at the final time $t_f = 80$, their sum and the percentage of predators over the total fish stock (preys and predators) in the sea, i.e.

$$R(t_f) = \frac{s(t_f)}{p(t_f) + s(t_f)}$$

(The function $R(t)$ was introduced on page 157.) □

The results of table 6.1 and Fig. 6.26 point out that when fishing is relaxed (lower b_d), prey numbers decline, while the number of predators, and so their percentage, rises.

6.5.5 The Prey-Predator Model with Harvest: The Validation

With the numerical solution of exercise *Hakes and sharks (2): fishing* we have validated the full model, i.e. the prey-predator model with harvest (6.14) introduced on page 188.

Table 6.2 Evolution of the species for the data of exercise *Hakes and sharks (2): fishing* when $b_d \in \{0.4, 0.5, 0.6\}$

b_d	Prey population	Predator population
0.4	Survival	Survival
0.5	Survival (bounded by carrying capacity)	**extinction**
0.6	**Extinction**	**Extinction**

Answering to D'Ancona's question

In particular, we have managed to answer the question posed by Umberto D'Ancona. The mathematical model we have formulated reflects the behaviour observed in the real world: a reduction in fishing causes a proliferation of predators over their prey.

That said, we should warn readers against drawing the wrong conclusions: since a decrease in fishing causes preys to decrease, one could be led to think that indiscriminate fishing would necessarily imply an increase of prey fish.

In reality things do not go that way, and our model reflects this phenomenon very well.

If we rerun the simulation with larger and larger depletion rates, say $b_d = 0.4$, $b_d = 0.5$, $b_d = 0.6$, we obtain the values of Table 6.2, which do not need any comment!

What we witness is a sustainable and beneficial situation for both the fishing market and the marine species, provided fishing is conducted with an eye to the dynamics of the populations (the natural growth rates, the environmental carrying capacity and the interaction between preys and predators).

Exercises

Solve Exercise 6.6, p. 231.

Mathematics as a virtual laboratory

We can also say the mathematical model we have constructed is a perfect tool for estimating (at a very low cost) how fishing may affect, for the better or the worse, a given marine ecosystem.

6.6 What We Have Learnt

The Way to Proceed

1. To construct a mathematical model we often have to simplify the real-world problem.

2. We need to isolate the actions to be studied from those we consider less important, and discard the latter.
3. The paradigm we should keep in mind is:

   - simplify in order to model;
   - validate the model;
   - enhance the model to get closer to reality.

Models

1. The *model of natural dynamics* (6.2) (a.k.a. Malthus law), introduced on page 160, describes the evolution of a population with unlimited food resources and no predators.
2. Fishing, or any type of harvesting, can be considered an *interaction with the surrounding environment*. The mathematical model (6.4) *of natural dynamics with harvest*, introduced on page 165, accounts for this phenomenon and is a variant of the natural dynamics model.
3. The *competition* for food and space within a population alters the natural dynamics and is modelled by the *logistic equation*, or *Verhulst equation*, (6.11), described on page 180. The mathematical model (6.12) *of a natural dynamics with harvest and competition*, introduced on page 180, accounts for the depletion due to fishing and the competition inside a species.
4. The *prey-predator* model (6.13), introduced on page 184, describes a system of two antagonist populations, but isolated from the environment.
5. The *prey-predator model with harvest* (6.14), introduced on page 188, describes a system of two species interacting with one another and with the environment (and as such it allows, in the case of fish populations, to account for fishing).

Mathematical Tools

1. A *differential equation* is a relationship between an unknown function and its derivatives. Some differential equations (not all) can be solved by analytical methods.
2. A *Cauchy problem* is a system consisting in a differential equation and an initial condition.
3. When a differential equation cannot be solved analytically, we can use numerical methods. These methods approximate derivatives by difference quotients and yield a numerical solution that approximates the exact solution.
4. The *Euler method* is a numerical method of order one (with respect to the discretisation step h) that approximates the solution of a Cauchy problem. When solving, a large number of points for computing the numerical solution (i.e. a small discretisation step h) yields a more accurate numerical solution, but also a greater computational complexity.

6.7 Looking Beyond

We started from a concrete problem (the question posed by Umberto D'Ancona) and we simplified it, focussing on the objects we wanted to describe and sweeping under the carpet other, more marginal, aspects.

The introduction of a number of simplifications at various stages allowed us to achieve a very simple model (differential problem (6.2) of page 160, describing the natural dynamics of one population with unlimited food and no predators), which we managed to solve by hand.

Then we began to make the model more realistic by reintroducing the various factors we had earlier neglected, in order to return to the initial concrete problem.

The final model (system (6.14), page 188, of two differential equations governing the interaction of two populations of preys and predators and the effects of fishing) gave us results in agreement with the experimental measurements found by D'Ancona.

In other words, we *validated* our mathematical model.

At this point we could consider adapting the mathematical model to other environments: no longer fish but, say, insects. We could for instance tackle the problem of the spread of Australian insects that were introduced in the US in 1868. What happened was that some bugs (Icerya purchasi) were accidentally brought in the country and threatened the cultivation of oranges. The US responded by introducing the natural predator of that species, the Rodolia Cardinalis ladybird, also Australian, which mostly solved the problem (see Fig. 6.27). But eventually, DDT had to be employed on a large scale to eradicate the infestation, resulting in a massive increase in the Icerya purchasi population (the prey) compared to ladybirds. The exact opposite of the original intention.

The model we have developed is able to describe very well this phenomenon, too. The insecticide affects the bugs just like fishing affects fish: by increasing the harvest (up to a certain threshold) the prey population grows.

We should point out that this model can be perfected, in the sense that it does describe a large number of phenomena, but it is not able to capture all possible interactions between the populations.

We could try to generalise the model to study systems of more than two populations, or more complex phenomena like the spread of an epidemic.

Fig. 6.27 Rodolia Cardinalis ladybird and Icerya purchasi. (Image Number: 5385575, Florida Division of Plant Industry, Florida Department of Agriculture and Consumer Services, Bugwood.org. Under Creative Commons Attribution 3.0 License)

Staying with fish, a next step to improve the model would be subsuming natural effects, such as sea currents dispersing plankton, and hence food. But such an extension would require an important conceptual effort. We would have to introduce functions p and s depending on t and also on the spatial coordinates x, y, z (four variables instead of one). We would need to talk about derivatives in the variables x, y, z, together with the t-derivatives. Eventually we would have to formulate a mathematical model involving *partial derivatives*.

6.8 List of Functions and Scripts of the Chapter

```
function [tn,un]=euler(f,t0,tf,y0,Nh)
% euler: Euler method for the Cauchy problem
% Calling instruction: [tn,un]=euler(f,t0,tf,y0,Nh)
% Approximation of the solution of Cauchy problem:
%                  y'(t)=f(t,y(t))   for t in [t0,tf]
%                  y(t0)=y0
% Input: f = function handle for the function f(t,y)
%        t0,tf = endpoints of time interval
%        y0 = datum of initial condition
%        Nh = number of points of discretisation (excluding t0)
% Output: tn = column vector of discretisation points
%              (including t0)
%         un = column vector of numerical solution
%              (including u0=y0)
tn=linspace(t0,tf,Nh+1)'; % discretisation points
h=(tf-t0)/Nh;              % discretisation step
un=zeros(Nh+1,1); % initialises the vector of the numerical
%                   solution
un(1)=y0; % stores the initial condition in the first
%           component of vector one
for n=1:Nh % computes the numerical solution by the Euler method
    un(n+1)=un(n)+h*f(tn(n),un(n));
end
```

Function 6.1: `euler.m`: the Euler method.

```
function cauchy_problem(Nh)
% cauchy_problem: solves the Cauchy problem by the Euler method
% Calling instruction: cauchy_problem(Nh)
% Input: Nh = number of time steps
% Output: graphic window

% define the data of the problem
t0=0; tf=10; % endpoints of time interval
y0=0;        % initial condition
f=@(t,y)(cos(t)-y)./(t+1); % function f(t,y)
% compute the numerical solution by the Euler method
[tn,un]=euler(f,t0,tf,y0,Nh);
% plot the numerical solution
figure(1);
plot(tn,un,'.','Markersize',16); grid on; hold on
xlabel('t'); ylabel('y(t)')
```

Function 6.2: `cauchy_problem.m`: Function for solving the Cauchy problem of page 174.

```
function [un_final]=dolphins2(Nh)
% dolphins2: solves problem "Aeolian dolphins (2)"
% Calling instruction: un_final=dolphins2(Nh)
% Input: Nh = number of time steps
% Output: un_final = solution at the final time

% define the data of the problem
t0=2017.5; tf=2025.5; % endpoints of time interval
y0=100;               % initial condition
% function f(t,y)
f=@(t,y)(0.06-0.03*(2-exp(-5*(sin(t*pi))^2)))*y;
% compute the numerical solution by the Euler method
[tn,un]=euler(f,t0,tf,y0,Nh);
un_final=un(end); % saves the numerical solution at time tf
% plot the numerical solution over the entire time interval
figure(1);
plot(tn,un,'.','Markersize',16); grid on; hold on
xlabel('t'); ylabel('y(t)')
```

Function 6.3: `dolphins2.m`: Function for solving exercise *Aeolian dolphins (2): the effect of the environment*.

```
function [un_final]=dolphins3(Nh)
% dolphins3: solves problem "Aeolian dolphins (3)"
% Calling instruction: un_final=dolphins3(Nh)
% Input: Nh = number of time steps
% Output: un_final = solution at the final time

% define the data of the problem
t0=2017.5; tf=2025.5; % endpoints of the time interval
y0=100;               % initial condition
b=@(t)-0.03*(2-exp(-5*(sin(t*pi)).^2)); % function b(t)
f=@(t,y)0.06*(1-y/150)*y+b(t)*y;       % function f(t,y)
% compute the numerical solution by the Euler method
[tn,un]=euler(f,t0,tf,y0,Nh);
un_final=un(end); % saves the numerical solution at time tf
% plot the numerical solution over the entire time interval
figure(1);
plot(tn,un,'.','Markersize',16); grid on
xlabel('t'); ylabel('y(t)')
```

Function 6.4: `dolphins3.m`: Function for solving exercise *Aeolian dolphins (3): the influence of the environment and competition.*

```
function [tn,un,vn]=euler2(f1,f2,t0,tf,p0,s0,Nh)
% euler2: Euler method for 2-equations Cauchy problem
% Calling instruction: [tn,un,vn]=euler2(f1,f2,t0,tf,p0,s0,Nh)
% Approximation of the solution of the Cauchy problem
% in two equations:
%             p'(t)=f1(t,p(t),s(t))   for t in [t0,tf]
%             s'(t)=f2(t,p(t),s(t))   for t in [t0,tf]
%             p(t0)=p0, s(t0)=s0
% Input:  f1,f2 = function handle of f1(t,p,s) and f2(t,p,s)
%         t0,tf = endpoints of interval
%         p0,s0 = initial conditions for p(t) and s(t) resp.
%         Nh    = number of points in discretisation (except t0)
% Output: tn    = vector of discretisation points (including t0)
%         un,vn = vectors of numerical solution
%                 (including u0=p0 and v0=s0)
tn=linspace(t0,tf,Nh+1)'; % discretisation points
h=(tf-t0)/Nh;             % discretisation step
un=zeros(Nh+1,1); % initialises the vectors of the numerical
%                   solution
vn=zeros(Nh+1,1);
un(1)=p0; vn(1)=s0; % saves initial conditions
for n=1:Nh % constructs the numerical solution by the
%            Euler method
    un(n+1)=un(n)+h*f1(tn(n),un(n),vn(n));
    vn(n+1)=vn(n)+h*f2(tn(n),un(n),vn(n));
end
```

Function 6.5: `euler2.m`: Euler method for a system of two differential equations.

```
function hakes_sharks(Nh)
% hakes_sharks: solves problem "Hakes and Sharks"
% Calling function: hakes_sharks(Nh)
% Input: Nh = number of time steps
% Output: graphic window

t0=0; tf=80; % endpoints of time interval
p0=5000; s0=40; % initial conditions
rp=.6; rs=-.2; Kp=3.e5; sigmap=rp/Kp; % data
sigmas=1.e-4; alpha=0.001; mu=.01;
f1=@(t,p,s)(rp-sigmap*p-alpha*s)*p;
f2=@(t,p,s)(rs-sigmas*s+mu*alpha*p)*s;
% compute the numerical solution by the Euler method
[tn,un,vn]=euler2(f1,f2,t0,tf,p0,s0,Nh);
fprintf('%6.0f hakes and %6.0f sharks at time t=%d \n',...
    un(end),vn(end),tf) % prints on screen
% plot the numerical solution
% the command plotyy uses two different scales on the same graph
figure(1); clf;
[ax,h1,h2]=plotyy(tn,un,tn,vn); grid on
set(h1,'Linewidth',2);
set(h2,'Linewidth',2);
% plot the phase-plane
figure(2); clf;
plot(un(1),vn(1),'r*',un,vn,'Linewidth',2);
grid on
```

Function 6.6: `hakes_sharks.m`: Function for solving the Cauchy problem of page 186.

```
function hakes_sharks_fishing(Nh)
% hakes_sharks_fishing: solves problem "Hakes and Sharks
% with fishing"
% Calling instruction: hakes_sharks_fishing(Nh)
% Input:  Nh = number of time steps
% Output: graphic window

% define the data of the problem
t0=0; tf=80; % endpoints of time interval
p0=5000; s0=40; % initial conditions
rp=.6; rs=-.2; Kp=3.e5; sigmap=rp/Kp; % data
sigmas=1.e-4; alpha=0.001; mu=.01;
figure(1); clf; figure(2); clf; % opens and wipes figures
for bd=[0.3,0.1,0] % loop over three different values of bd
f1=@(t,p,s)(rp-sigmap*p-alpha*s-bd)*p;
f2=@(t,p,s)(rs-sigmas*s+mu*alpha*p-bd)*s;
% compute the numerical solution by the Euler method
[tn,un,vn]=euler2(f1,f2,t0,tf,p0,s0,Nh);
fprintf('bd=%2.1f, %6.0f hakes and %6.0f sharks at t=%d \n',...
         bd,un(end),vn(end),tf) % prints on screen
% plot the numerical solution
figure(1);
plot(tn,un,'Linewidth',2); hold on
figure(2);
plot(tn,vn,'Linewidth',2); hold on
end
figure(1); grid on;
xlabel('t'); ylabel('preys (hakes)');
legend('bd=0.3','bd=0.1','bd=0')
figure(2); grid on;
xlabel('t'); ylabel('predators (sharks)');
legend('bd=0.3','bd=0.1','bd=0')
hold off
```

Function 6.7: `hakes_sharks_fishing.m`: Function for solving the Cauchy problem of page 189.

```
% script finite_diff1
% approximation of f'(t) by finite differences
f=@(t)sin(t); % defines function f(t)
tc=pi/6;       % point at which to compute the approximations
de=cos(pi/6); % exact value of derivative at tc=pi/6
% H= vector of discretisation parameters
H=[0.1,0.05,0.01,0.005,0.001,0.0005,0.0001];
for h=H
    df=(f(tc+h)-f(tc))/h;  % finite difference
    fprintf('f''(tc)=%f, h=%f, df=%f \n',de,h,df)  % prints
end
```

Script 6.8: `finite_diff1.m`: Approximation of the first derivative by forward finite differences.

```
% script finite_diff2
% approximation of f'(t) by finite differences - errors
f=@(t)sin(t); tc=pi/6; de=cos(pi/6);
% H= vector of discretisation parameters
H=[0.1,0.05,0.01,0.005,0.001,0.0005,0.0001];
for h=H
    df=(f(tc+h)-f(tc))/h; %  finite difference
%   abs(x) computes the absolute value of x
    err=abs(de-df); % error.
    fprintf('h=%f, err=%e \n',h,err) % prints the error
end
```

Script 6.9: `finite_diff2.m`: Computation of errors of forward finite differences.

```
% script finite_diff3
% approximation of f'(t) by finite differences - errors - plot
f=@(t)sin(t); tc=pi/6; de=cos(pi/6);
% H= vector of discretisation parameters
H=[0.1,0.05,0.01,0.005,0.001,0.0005,0.0001];
Err=[ ]; % initialises an empty vector;
for h=H
    df=(f(tc+h)-f(tc))/h; % finite difference
    err=abs(de-df); % error
    Err=[Err, err]; % error vector
end
figure(1); clf
loglog(H,Err,'bo--',H,H,'k-'); % plots, logarithmic scale
grid on
xlabel('h','Fontsize',20); ylabel('Errors','Fontsize',20)
le=legend('|f''(t_c)-df|','h'); % defines the legend
% position of legend
set(le,'Fontsize',20,'Location','Southeast');
set(gca,'Fontsize',20) % font for the whole picture
```

Script 6.10: `finite_diff3.m`: Plot of the errors of forward finite differences.

7 Take-Home Message

Let us try to gather our thoughts. What did we learn?

We have learnt that a mathematical model of a physical (or real) problem requires a good knowledge of the problem itself. Often – almost always – this knowledge must be shared with the experts on the problem, be these engineers, biologists, economists, physicists or physicians.

We have discovered that a mathematical model is fed by a set of data (generated by the real problem) and provides a solution that characterises the behaviour of the problem.

We have understood that to a given physical problem there may correspond several models. This should not surprise us that much. Modelling almost invariably requires that we simplify, by making assumptions meant to reduce the complexity of the initial problem without betraying its essence. Depending on the simplifying hypotheses made, we will find a corresponding mathematical model.

We have also discovered that this "multi-valued correspondence" is actually "invertible", because a specific mathematical problem may be used as a model for rather different physical problems. The meaning of the unknowns and of the data clearly changes from problem to problem, but the "structure" of the mathematical problem stays the same.

This feature is truly extraordinary and highlights the great encompassing power of mathematical models: they are able to bring into focus diverse physical processes using the same equations.

We have learnt that very often – practically always – a mathematical model cannot be solved by hand, since there is no magic formula to represent its solution. For this reason a mathematical model is paired with a numerical model. The latter can always be solved through an algorithm implementable on the computer by a programming language. We have further observed that to handle a numerical model several algorithms may be employed, each one characterised by a different computational cost (memory space, computing time).

This process, which starts from the mathematical model and ends with the computer-generated results, is the realm of Scientific Computing.

© Springer Nature Switzerland AG 2020

A. Quarteroni, P. Gervasio, *A Primer on Mathematical Modelling*, UNITEXT 121,
https://doi.org/10.1007/978-3-030-44541-6_7

8 Solutions to Exercises

8.1 Exercises of Chapter 2

Exercise 2.1. Given the function $f(x) = -x^2 + 4x + 1$, construct f_{mp} over the interval $[1, 4]$ using a partition made of $M = 6$ subintervals of the same length. Plot f and f_{mp} on the same plane.

Solution. The step function f_{mp} assumes, over the interval I_k, the constant value $f(\overline{x}_k)$, for $k = 1, \ldots, M$, where $\overline{x}_k$ is the midpoint of I_k.

We start by computing the partition points $x_0, \ldots, x_M$, then the midpoints $\overline{x}_1, \ldots \overline{x}_M$ and finally the values $f(\overline{x}_1), \ldots, f(\overline{x}_M)$:

$$x_0 = 1,\ x_1 = \frac{3}{2},\ x_2 = 2,\ x_3 = \frac{5}{2},\ x_4 = 3,\ x_5 = \frac{7}{2},\ x_6 = 4,$$

$$\overline{x}_1 = \frac{5}{4},\ \overline{x}_2 = \frac{7}{4},\ \overline{x}_3 = \frac{9}{4},\ \overline{x}_4 = \frac{11}{4},\ \overline{x}_5 = \frac{13}{4},\ \overline{x}_6 = \frac{15}{4},$$

$$f(\overline{x}_1) = \frac{71}{16},\ f(\overline{x}_2) = \frac{79}{16},\ f(\overline{x}_3) = \frac{79}{16},\ f(\overline{x}_4) = \frac{71}{16},\ f(\overline{x}_5) = \frac{55}{16},\ f(\overline{x}_6) = \frac{31}{16}.$$

Now we can plot the functions $f(x)$ and $f_{mp}(x)$, whose graphs appear in Fig. 8.1. □

Exercise 2.2. Given the function $f(x) = -x^2 + 4x + 1$ defined over the interval $[1, 4]$, compute $area(T_{mp})$, using $M = 6$ subintervals of the same length.

Solution. Let us recall the midpoint quadrature formula:

$$area(T_{mp}) = h \sum_{k=1}^{M} f(\overline{x}_k).$$

© Springer Nature Switzerland AG 2020

A. Quarteroni, P. Gervasio, *A Primer on Mathematical Modelling*, UNITEXT 121,
https://doi.org/10.1007/978-3-030-44541-6_8

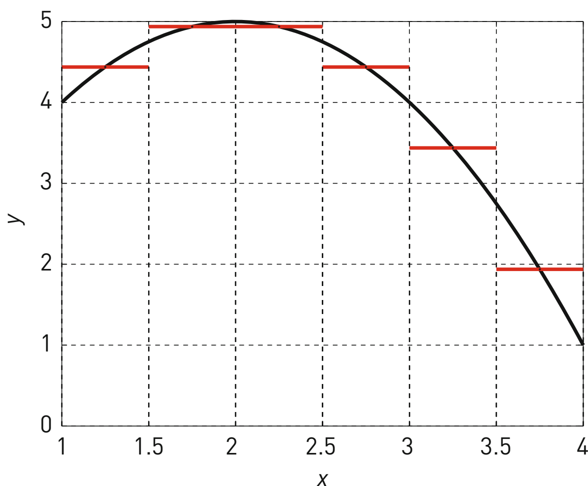


Fig. 8.1 The functions $f(x)$ (black) and $f_{mp}(x)$ (red)

Using the numerical values computed for the previous exercise we can determine $h = \dfrac{b-a}{M} = \dfrac{1}{2}$. So we have:

$$area(T_{mp}) = \frac{1}{2}\left[\frac{71}{16} + \frac{79}{16} + \frac{79}{16} + \frac{71}{16} + \frac{55}{16} + \frac{31}{16}\right]$$

$$= \frac{193}{16} = 12.0625.$$

This value represents a very good approximation of the exact area, which is equal to 12. □

Exercise 2.3. Given the matrix $A = \begin{bmatrix} 4 & 1.5 & 2.5 \\ 1 & 1 & 1 \\ -1 & -2 & 1 \end{bmatrix}$ and the vector $\mathbf{c} = \begin{bmatrix} 2 \\ -1 \\ 3 \end{bmatrix}$, compute the matrix-vector product $A\mathbf{c}$.

Solution. By using formula (2.8) on page 35, we must build the vector

$$A\mathbf{c} = \begin{bmatrix} a_{11}c_1 + a_{12}c_2 + a_{13}c_3 \\ a_{21}c_1 + a_{22}c_2 + a_{23}c_3 \\ a_{31}c_1 + a_{32}c_2 + a_{33}c_3 \end{bmatrix}.$$

So:

$$A\mathbf{c} = \begin{bmatrix} 4 & 1.5 & 2.5 \\ 1 & 1 & 1 \\ -1 & -2 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ -1 \\ 3 \end{bmatrix} = \begin{bmatrix} 4 \cdot 2 + 1.5 \cdot (-1) + 2.5 \cdot 3 \\ 1 \cdot 2 + 1 \cdot (-1) + 1 \cdot 3 \\ -1 \cdot 2 + (-2) \cdot (-1) + 1 \cdot 3 \end{bmatrix}$$

$$= \begin{bmatrix} 14 \\ 4 \\ 3 \end{bmatrix}$$

□

Exercise 2.4. Write a 4×4 matrix A and a column vector $\mathbf{c}$ with four components, then compute their product.

Solution. We choose $A = \begin{bmatrix} 2 & 0 & -3 & 5 \\ -1.5 & -1 & 2.5 & 0 \\ 4 & -2 & 1 & 0 \\ -1 & 3 & 2 & 3 \end{bmatrix}$ and $\mathbf{c} = \begin{bmatrix} -2 \\ 0 \\ -1 \\ 3 \end{bmatrix}$.

By formula (2.8), page 35, we have to build the vector

$$A\mathbf{c} = \begin{bmatrix} a_{11}c_1 + a_{12}c_2 + a_{13}c_3 + a_{14}c_4 \\ a_{21}c_1 + a_{22}c_2 + a_{23}c_3 + a_{24}c_4 \\ a_{31}c_1 + a_{32}c_2 + a_{33}c_3 + a_{34}c_4 \\ a_{41}c_1 + a_{42}c_2 + a_{43}c_3 + a_{44}c_4 \end{bmatrix}.$$

Hence:

$$A\mathbf{c} = \begin{bmatrix} 2 & 0 & -3 & 5 \\ -1.5 & -1 & 2.5 & 0 \\ 4 & -2 & 1 & 0 \\ -1 & 3 & 2 & 3 \end{bmatrix} \begin{bmatrix} -2 \\ 0 \\ -1 \\ 3 \end{bmatrix}$$

$$= \begin{bmatrix} 2 \cdot (-2) + 0 \cdot 0 + (-3) \cdot (-1) + 5 \cdot 3 \\ (-1.5) \cdot (-2) + (-1) \cdot 0 + 2.5 \cdot (-1) + 0 \cdot 3 \\ 4 \cdot (-2) + (-2) \cdot 0 + 1 \cdot (-1) + 0 \cdot 3 \\ (-1) \cdot (-2) + 3 \cdot 0 + 2 \cdot (-1) + 3 \cdot 3 \end{bmatrix} = \begin{bmatrix} 14 \\ 0.5 \\ -9 \\ 9 \end{bmatrix}.$$

□

Exercise 2.5. Solve the linear system

$$\begin{cases} x_1 + x_3 = 3 \\ 2x_1 + 2x_2 - x_3 = -4 \\ 3x_1 - 6x_2 - 4x_3 = 7 \end{cases}$$

by following the instructions of Algorithm 2, page 44.

Solution. Let us follow the instructions of Algorithm 2, page 44, and execute the loop over k. Call R_1 the first equation of the system (R is for row), R_2 the second equation and R_3 the third.

Set $k = 1$. The index i (which varies between $k + 1$ and 3) can be 2 or 3 (first 2, then 3), while the index j assumes all values between 1 and 3:

set $k = 1$

set $i = 2$

– compute the multiplier $m_{ik} = m_{21} = \frac{2}{1} = 2$;

– determine the new row R_2 by computing the algebraic sum $R_2 - m_{21} \cdot R_1$ term by term:

$$\begin{array}{rr} R_2: & 2x_1 + 2x_2 - x_3 = -4 \\ -m_{21}R_1: & -2 \cdot (x_1 + 0x_2 + x_3 = 3) \\ \hline \text{new row } R_2: & 0x_1 + 2x_2 - 3x_3 = -10 \end{array}$$

set $i = 3$

– compute the multiplier $m_{ik} = m_{31} = \frac{3}{1} = 3$;

– determine the new row R_3 by computing the algebraic sum $R_3 - m_{31} \cdot R_1$ term by term:

$$\begin{array}{rr} R_3: & 3x_1 - 6x_2 - 4x_3 = 7 \\ -m_{31}R_1: & -3 \cdot (x_1 + 0x_2 + x_3 = 3) \\ \hline \text{new row } R_3: & 0x_1 - 6x_2 - 7x_3 = -2 \end{array}$$

Now, since the instructions for $k = 1$ are finished, we rewrite the updated system, where the original second and third equations are replaced by the ones just found, namely

$$\begin{cases} x_1 + x_3 = 3 \\ 2x_2 - 3x_3 = -10 \\ -6x_2 - 7x_3 = -2. \end{cases} \tag{8.1}$$

Then we continue executing our algorithm on the latter system, by taking $k = 2$.

Set $k = 2$

set $i = 3$

– compute the multiplier $m_{ik} = m_{32} = -\frac{6}{2} = -3$;

– compute the new row R_3 by computing the algebraic sum $R_3 - m_{32} \cdot R_2$ term by term:

$$\begin{array}{rr} R_3: & -6x_2 - 7x_3 = -2 \\ -m_{32}R_2: & -(-3) \cdot (2x_2 - 3x_3 = -10) \\ \hline \text{new row } R_3: & 0x_2 - 16x_3 = -32 \end{array}$$

As the instructions for $k = 2$ have ended, we rewrite the system by replacing the third equation with the one obtained above:

$$\begin{cases} x_1 + x_3 = 3 \\ 2x_2 - 3x_3 = -10 \\ -16x_3 = -32. \end{cases} \tag{8.2}$$

The reduction phase is concluded. The last equation contains just one unknown, so we may isolate it and pass to the back-substitution of the variables (starting from the last equation, backwards, up to the first one).

Compute x_3 from the last equation of system (8.2): $x_3 = \dfrac{-32}{-16} = 2$;

(now we have to execute a loop on the index $i = 2, 1$);

set $i = 2$

– isolate x_2 and substitute x_3 in the second equation of system (8.2):

$$x_2 = \frac{-10 + 3x_3}{2} = \frac{-10 + 6}{2} = -2,$$

set $i = 1$

– isolate x_1 and substitute x_2 and x_3 in the first equation of system (8.2), that is:

$$x_1 = \frac{3 - 0x_2 - x_3}{1} = 3 - 2 = 1.$$

The solution to the system is $x_1 = 1, x_2 = -2, x_3 = 2$. □

Exercise 2.6. How long would it take a personal computer performing 10^7 elementary operations per second to solve a linear system of dimension $n = 100$ using the Cramer method?

Solution. We saw that the Cramer method on a linear system of dimension n requires roughly $3(n + 1)!$ elementary operations. So for $n = 100$ we have $3(100 + 1)! \sim 3 \cdot 10^{160}$. To find out how much time it would take a computer that does 10^7 elementary operations per second, we divide $3 \cdot 10^{160}$ by 10^7.

Our personal computer would roughly take $3 \cdot 10^{153}$ seconds, corresponding to $5 \cdot 10^{151}$ minutes, or $8.\bar{3} \cdot 10^{149}$ hours, or $3.5 \cdot 10^{148}$ days. In other words, $9.5 \cdot 10^{145}$ years! □

Exercise 2.7. Given the linear system

$$\begin{cases} x_1 + x_3 = 3 \\ 2x_1 + 2x_2 - x_3 = -4 \\ 3x_1 - 6x_2 - 4x_3 = 7 \end{cases},$$

write it in matrix form and solve it using the instructions in Algorithm 3, page 49. Compare the solution method with that of Exercise 2.5.

Solution. The coefficients of the unknowns x_1, x_2, x_3 in the first equation are 1, 0 (as x_2 does not appear in the equation) and 1, so these are the elements of the first row of matrix A. Similarly, from the second row we extract the coefficients 2, 2, -1, and from the third row we get 3, -6 and -4. The matrix A associated with the system therefore reads:

$$A = \begin{bmatrix} 1 & 0 & 1 \\ 2 & 2 & -1 \\ 3 & -6 & -4 \end{bmatrix}.$$

The vector $\mathbf{b}$ contains (in the same order) the right-hand sides of the equations, 3, -4 and 7, whereas the solution vector contains the unknowns x_1, x_2, x_3:

$$\mathbf{b} = \begin{bmatrix} 3 \\ -4 \\ 7 \end{bmatrix}, \qquad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}.$$

Then we execute the operations indicated by Algorithm 3, which we shall copy here replacing n by the value 3:

```
% reduction of the unknowns
for k = 1,...,2 do
    for i = k + 1,...,3 do
        m_ik = a_ik/a_kk;
        for j = 1,...,3 do
            a_ij = a_ij − m_ik a_kj;
        end
        b_i = b_i − m_ik b_k;
    end
end
% back-substitution of the unknowns
x_3 = b_3/a_33;
for i = 2,1 do
    s = 0;
    for j = i + 1,...,3 do
        s = s + a_ij x_j;
    end
    x_i = (b_i − s)/a_ii;
end
```

Let us examine the reduction phase, where we have three nested loops: first a loop on k, inside of which there is a loop on i, inside of which there is a further loop on j (it's like having three boxes one inside the other).

During the reduction we did not replace the *for loop* $j = 1, \ldots, n$ with the *for loop* $j = k+1, \ldots, n$, as suggested on page 49. In this exercise, in fact, we seek not only the unknowns, but we also want to see how the matrix A associated with the system varies. Hence we use an algorithm that computes all matrix entries, and not just the ones needed to find the solutions.

Fix $k = 1$. The index i (varying between $k + 1$ and 3) can take the values 2 and 3 (first one, then the other), while j can be anything between 1 and 3:

set $k = 1$

set $i = 2$

compute $m_{ik} = a_{ik}/a_{kk}$, with $k = 1$ and $i = 2$, i.e. compute $m_{21} = a_{21}/a_{11} = 2/1 = 2$;

set $j = 1$

compute $a_{ij} - m_{ik}a_{kj}$ with $k = 1, i = 2$ and $j = 1$ and save it in a_{ij} (the element of A on row i and column j) in place of the previous value, i.e. compute $a_{21} - m_{21}a_{11} = 2 - 2 \cdot 1 = 0$ and save it in a_{21};

set $j = 2$

compute $a_{ij} - m_{ik}a_{kj}$ with $k = 1, i = 2$ and $j = 2$ and save it in a_{ij} in place of the old value, i.e. compute $a_{22} - m_{21}a_{12} = 2 - 2 \cdot 0 = 2$ and save it in a_{22};

set $j = 3$

compute $a_{ij} - m_{ik}a_{kj}$ with $k = 1, i = 2$ and $j = 3$ and save it in a_{ij} in place of the old value, i.e. compute $a_{23} - m_{21}a_{13} = -1 - 2 \cdot 1 = -3$ and save it in a_{23};

we have exhausted all possible values for j

compute $b_i - m_{ik}b_k$ with $k = 1, i = 2$ and save it in component i of vector $\mathbf{b}$ in place of the old value, i.e. compute $b_2 - m_{21}b_1 = -4 - 2 \cdot 3 = -10$ and save it in b_2.

Now we are done with $i = 2$, so we move on to the next value $i = 3$, and repeat with $j = 1, 2, 3$ (and henceforth we shall shorten the explanations, since they do not change). We have:

(k always equals 1*)*

set $i = 3$

compute $m_{ik} = a_{ik}/a_{kk}$ with $k = 1, i = 3$, i.e. $m_{31} = a_{31}/a_{11} = 3/1 = 3$;

set $j = 1$

compute $a_{ij} - m_{ik}a_{kj}$ with $k = 1, i = 3$ and $j = 1$ and save it in a_{ij} in place of the old value, i.e. compute $a_{31} - m_{31}a_{11} = 3 - 3 \cdot 1 = 0$ and save it in a_{31};

set $j = 2$

compute $a_{ij} - m_{ik}a_{kj}$ with $k = 1, i = 3$ and $j = 2$ and save it in a_{ij} in place of the old value, i.e. compute $a_{32} - m_{31}a_{12} = -6 - 3 \cdot 0 = -6$ and save it in a_{32};

set $j = 3$

compute $a_{ij} - m_{ik}a_{kj}$ with $k = 1, i = 3$ and $j = 3$ and save it in a_{ij} in place of the old value, i.e. compute $a_{33} - m_{31}a_{13} = -4 - 3 \cdot 1 = -7$ and save it in a_{33};

there are no more choices left for j

compute $b_i - m_{ik}b_k$ with $k = 1, i = 3$ and save it in component i of vector $\mathbf{b}$ in place of the old value, i.e. compute $b_3 - m_{31}b_1 = 7 - 3 \cdot 3 = -2$ and save it in b_3;

this ends the work for $i = 3$.

We have gone through all possible values of j and i corresponding to $k = 1$, so we have finished with $k = 1$.

As we have modified some elements in A and $\mathbf{b}$, before we proceed with $k = 2$ we rewrite the matrix and the right-hand-side vector with the updated entries found above:

$$A = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 2 & -3 \\ 0 & -6 & -7 \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} 3 \\ -10 \\ -2 \end{bmatrix}. \tag{8.3}$$

Note that the elements in the first column of A below the diagonal, a_{21} and a_{31}, are now zero. This means we have eliminated the unknown x_1 from the second and third equations.

Let us repeat everything for $k = 2$, using A and $\mathbf{b}$ from (8.3), and observe that now i can only be 3. We have:

set $k = 2$
- *set* $i = 3$
  - compute $m_{ik} = a_{ik}/a_{kk}$ for $k = 2, i = 3$, i.e. $m_{32} = a_{32}/a_{22} = -6/2 = -3$;
  - *set* $j = 1$
    - compute $a_{ij} - m_{ik}a_{kj}$ with $k = 2, i = 3$ and $j = 1$ and save it in a_{ij} in place of the old value, i.e. compute $a_{31} - m_{32}a_{21} = 0 - (-3) \cdot 0 = 0$ and save it in a_{31};
  - *set* $j = 2$
    - compute $a_{ij} - m_{ik}a_{kj}$ with $k = 2, i = 3$ and $j = 2$ and save it in a_{ij} in place of the old value, i.e. compute $a_{32} - m_{32}a_{22} = -6 - (-3) \cdot 2 = 0$ and save it in a_{32};
  - *set* $j = 3$
    - compute $a_{ij} - m_{ik}a_{kj}$ with $k = 2, i = 3$ and $j = 3$ and save it in a_{ij} in place of the old value, i.e. compute $a_{33} - m_{32}a_{23} = -7 - (-3) \cdot (-3) = -16$ and save it in a_{33};
  - *this exhausts all values for* j
  - compute $b_i - m_{ik}b_k$ with $k = 2, i = 3$ and save it in component i of $\mathbf{b}$ in place of the old value, i.e. compute $b_3 - m_{32}b_2 = -2 - (-3) \cdot (-10) = -32$ and save it in b_3.

At this juncture we have gone through all values for j and i corresponding to $k = 2$, so the work for $k = 2$ is done.

Rewriting A and $\mathbf{b}$ after the $k = 2$ phase,

$$A = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 2 & -3 \\ 0 & 0 & -16 \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} 3 \\ -10 \\ -32 \end{bmatrix}. \tag{8.4}$$

The second unknown has been eliminated from equation three.

Now the matrix A has been reduced to *upper triangular* form, meaning that the possible non-zero elements only live on the main diagonal or above it. The reduction part ends.

This is where the back-substitution begins, namely the concrete computation of the unknowns. We have to use the values of the elements a_{ij} in A and of the components b_i of $\mathbf{b}$ appearing in (8.4). By following the instructions of the algorithm, we have

compute $x_3 = b_3/a_{33} = -32/(-16) = 2$;
set $i = 2$
- define $s = 0$;
- the index j can assume only value 3, so we compute $s + a_{ij}x_j$ with $i = 2$ and $j = 3$, i.e. $s + a_{23}x_3 = 0 + (-3)\cdot 2 = -6$ and we save it in the variable s, so $s = -6$;
- compute $x_i = (b_i - s)/a_{ii}$ for $i = 2$, i.e. $x_2 = (b_2 - s)/a_{22} = (-10 - (-6))/2 = -2$

set $i = 1$
- define $s = 0$;
- the index j can assume values 2 and 3;
- *set* $j = 2$
  - compute $s + a_{ij}x_j$ with $i = 1$ and $j = 2$, i.e. $s + a_{12}x_2 = 0 + 0\cdot(-6) = 0$ and save it in the variable s in place of the old value, so again $s = 0$
- *set* $j = 3$
  - compute $s + a_{ij}x_j$ with $i = 1$ and $j = 3$, i.e. $s + a_{13}x_3 = 0 + 1\cdot 2 = 2$ and save it in the variable s in place of the old value, whence $s = 2$;
  - the loop on j has ended
- compute $x_i = (b_i - s)/a_{ii}$ with $i = 1$, i.e. $x_1 = (b_1 - s)/a_{11} = (3 - 2)/1 = 1$.

Now that the substitution has ended, we conclude that the solution to the system is

$$x_1 = 1, \qquad x_2 = -2, \qquad x_3 = 2.$$

Notice that the matrix and the right-hand side in (8.3) correspond to system (8.1), while those in (8.4) correspond to system (8.2). This confirms that Algorithms 2 and 3 are equivalent: the former acts on the system of equations, the latter on its matrix form. □

8.2 Exercises of Chapter 3

Exercise 3.1. Write a function that, fed with a square matrix A of order 2 or 3 as input, calculates its determinant using rules (2.5) and (2.9).

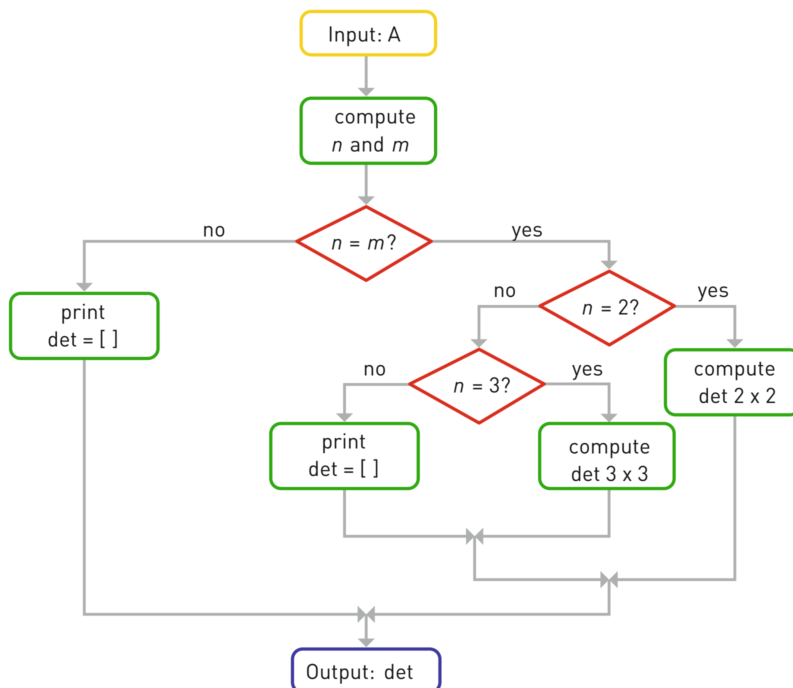
Solution. Starting from the sole variable `A` containing the square matrix A, our function must be capable to determine the number of rows and columns of the matrix, decide whether it is square and then find its order. Depending on the answers to this query on the matrix, the function should implement the correct formula for the determinant, meaning formula (2.5) if the matrix is square and 2×2, and (2.9) if

it is square and 3×3. In case the matrix is not square we want to display on screen "The matrix is not square", while if it is square but of dimension $n > 3$, we want "This function operates only with 2×2 or 3×3 matrices" to appear.

size

To determine the number n of rows and the number m of columns of an array we use the command `size(A)`. It gives two integer values as output, namely the number of rows and columns of array `A`, in that order.

To diversify the work, we have to use selection blocks, and to be precise in this fashion:



The two selection blocks with the questions "$n = 2$?" and "$n = 3$?" are nested and can be implemented using the condition with many alternatives (*if* ... *elseif* .... *else* .... *end*) that we have seen in Sect. 3.1.5.

Let us open the editor window and type the following instructions:

```
function [det]=det23(A)
% det23: computes the determinant of a 2x2 or 3x3 matrix
% Calling instruction:  d=det23(A)
% Input:  A    = matrix
% Output: det = the determinant of A, only if A is 2x2 or 3x3
%               otherwise a message of error is printed
[n,m]=size(A);  % computes the size of A
if n==m  %  det(A) is defined only if A is square
% check the dimension of A and apply the formula
  if n==2  % 2x2 case
    det = A(1,1)*A(2,2)-A(2,1)*A(1,2);
  elseif n == 3  %  3x3 case
    det = A(1,1)*(A(2,2)*A(3,3)-A(2,3)*A(3,2))...
          -A(1,2)*(A(2,1)*A(3,3)-A(2,3)*A(3,1))...
          +A(1,3)*(A(2,1)*A(3,2)-A(3,1)*A(2,2));
```

```
  else % means the size of A is larger than 3
    disp('This function operates only with 2x2 or 3x3 matrices');
    det=[ ];
  end
else
  disp('The matrix is not square');
  det=[ ];
end
```

Let us emphasise that the three dots `...` at the end of a line tell Octave the instruction is still not complete, and to continue reading it on the next line.

`...`

Then we store the function under the name `det23` and check it is correct by calling it up in the command window.

Let us start by storing a 2×2 matrix A in the variable `A`, call the `det23` function by giving in input the variable `A` and saving the final result in `detA`. The instructions read:

```
A=[2 4; -1 3]; % defines the variable A to be given as input
detA=det23(A)  % calls up the function
```

and Octave produces

```
detA =   10
```

Next, we save a 3×3 matrix B in the variable `B` and call up the `det23` function by giving `B` as input, then saving the result in `detB`. The instructions are:

```
B=[3 -1 3; 2 0 5; -4 2 1]; % defines the variable B (input)
detB=det23(B)  % calls up the function
```

and Octave gives

```
detB =   4
```

We leave it to the reader to check what the function does in case $n \neq m$, or when we give in input a matrix of order larger than 3.

Let us remind that when the function has only one output variable, the square brackets around the unique output are optional.

To compute the determinant of a square $n \times n$ matrix `A` (with n possibly larger than 3) we may use the Octave command `det(A)`. For example, the Octave instructions to compute the determinant of

`det`

$$A = \begin{bmatrix} 1 & 3 & 5 & -2 \\ 0 & 2 & -1 & 2 \\ 3 & 4 & 1 & 0 \\ -1 & 1 & 5 & 3 \end{bmatrix}$$

are:

```
A=[1, 3, 5, -2; 0, 2, -1, 2; 3, 4, 1, 0; -1, 1, 5, 3];
d=det(A)
```

and the result Octave gives is:

```
d = -165.00
```

□

Exercise 3.2. Write a function that implements the GEM of Algorithm 3, page 49.

Solution. Let us convert each line of Algorithm 3 into a corresponding Octave instruction. The inputs are the data of the algorithm, namely the matrix A and the right-hand-side vector $\mathbf{b}$, whilst the output is the result, i.e. the solution $\mathbf{x}$.

Before we actually do this translation, let us check in the function that the matrix is square and $\mathbf{b}$ has dimension equal to the order of A, in analogy to what we did in Exercise 3.1. If these conditions are not met, the function will not execute the operations and will display an alert saying the data are incorrect.

The required function is written at the end of the exercise.

Observe that the multiplier m_{ik} is the result of a division. Every time the computer has to divide, it is advisable to make sure that the denominator is different from 0. The block of instructions

```
if A(k,k)==0
warning ('GEM cannot terminate');
x =[ ]; return
end
```

serves exactly this purpose. In case $a_{kk} = 0$, we display a message by the instruction `warning`, we initialise the output variable `x` with an empty vector and we force Octave to terminate the execution of the function by the instruction `return`.

warning
return

When writing a function we must also check that it will produce the correct answers: in the jargon, we have to "test it". We shall test the `gem` function on

$$\begin{cases} x_1 + x_3 = 3 \\ 2x_1 + 2x_2 - x_3 = -4 \\ 3x_1 - 6x_2 - 4x_3 = 7 \end{cases} .$$

If we type the instructions

```
A=[1 0 1; 2 2 -1; 3 -6 -4];
b=[3; -4; 7];
x=gem(A,b)
```

Octave answers

```
x =
    1
   -2
    2
```

i.e. $x_1 = 1$, $x_2 = -2$ and $x_3 = 2$. To see whether the solution found is correct, we substitute the above values in the equations of the system and check that we get three identities.

The `gem` function will be useful once again in Chap. 5. □

```
function [x]=gem(A,b)
% gem: computes the solution to system Ax=b using GEM
% Calling instruction: x=gem(A,b)
% Input: A = square matrix of the system
%        b = column vector, right-hand-side term of the system
% Output: x = column vector, solution of the system
[n,m]= size(A); % computes the size of A
[nrb,ncb]=size(b); % computes the dimension of  b
if n == m && n==nrb
% the system is solved only if the matrix A is square and
% b is a column vector of the same dimension as A
% reduction step
  for k=1:n-1
     for i=k+1:n
        if A(k,k)==0
           warning('GEM cannot terminate');
           x=[ ]; return
        end
        mik=A(i,k)/A(k,k);
        for j=k+1:n
           A(i,j)=A(i,j)-mik*A(k,j);
        end
        b(i)=b(i)-mik*b(k);
     end
  end
% back-substitution step
  x=zeros(n,1);
  x(n)=b(n)/A(n,n);
  for i=n-1:-1:1
     s=0;
     for j=i+1:n
        s=s+A(i,j)*x(j);
     end
     x(i)=(b(i)-s)/A(i,i);
  end
else %  the system cannot be solved by GEM
disp('The matrix is not square')
x=[ ];
end
```

Function 8.1: `gem.m`: GEM (Gauss Elimination Method)

Exercise 3.3. Solve the linear system

$$\begin{cases} 2x_1 - 3x_2 + 5x_4 = -11.5 \\ -1.5x_1 - x_2 + 2.5x_3 = -3 \\ 4x_1 - 2x_2 + x_3 - x_4 = -18 \\ -x_1 + 3x_2 + 2x_3 + 3x_4 = 10 \end{cases}$$

in Octave by calling the function written for Exercise 3.2.

Solution. First of all let us write the expressions of the matrix A and the right-hand-side vector $\mathbf{b}$ associated with the given linear system:

$$A = \begin{bmatrix} 2 & -3 & 0 & 5 \\ -1.5 & -1 & 2.5 & 0 \\ 4 & -2 & 1 & -1 \\ -1 & 3 & 2 & 3 \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} -11.5 \\ -3 \\ -18 \\ 10 \end{bmatrix}.$$

Then we define the input data for the `gem` function (that is, the variable `A` containing A and the variable `b` containing the column vector $\mathbf{b}$) and call it up, saving in the variable `x` the result:

```
A=[2 -3 0 5; -1.5 -1 2.5 0; 4 -2 1 -1; -1 3 2 3];
b=[-11.5; -3; -18; 10];
x=gem(A,b)
```

This gives

```
x =
  -2.50000
   3.00000
  -1.50000
   0.50000
```

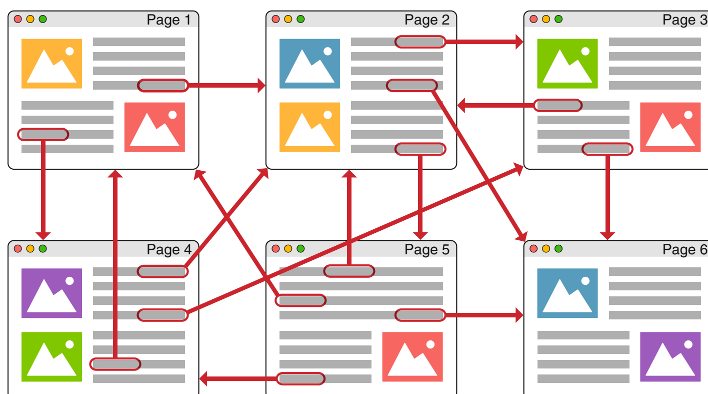
i.e.

$$x_1 = -2.5, \quad x_2 = 3, \quad x_3 = -1.5, \quad x_4 = 0.5.$$

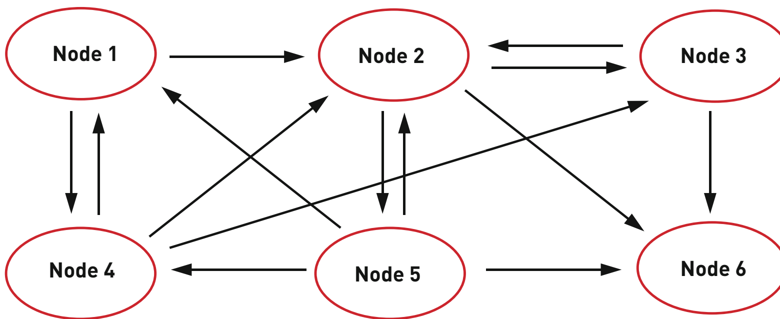
In order to check that the solution is correct, it is enough to multiply A by the solution vector $\mathbf{x}$ and verify the result coincides with $\mathbf{b}$. □

8.3 Exercises of Chapter 4

Exercise 4.1. Build the directed graph and the adjacency matrix associated with the following network:



Solution. The graph associated with the network is:



The adjacency matrix reads

$$A = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \end{bmatrix}$$

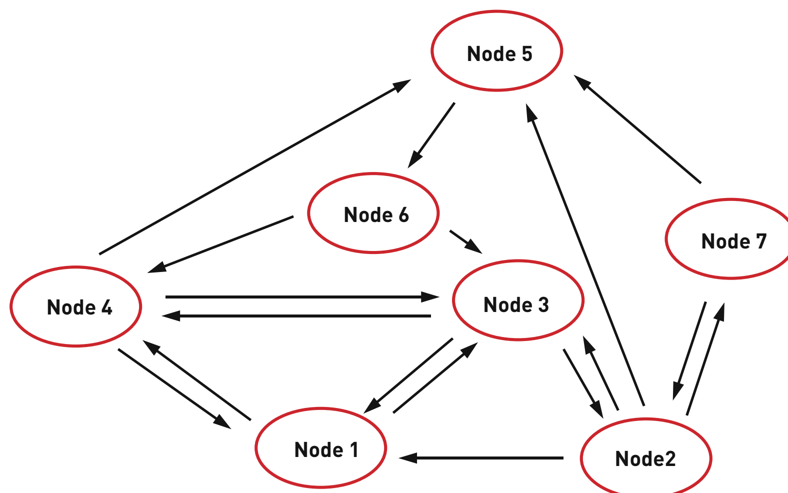
□

Exercise 4.2. Let

$$A = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

be the adjacency matrix of a directed graph. Sketch the graph.

Solution. The graph associated with the given matrix is



□

Exercise 4.3. Compute the PageRanks of the pages in the network of Exercise 4.1. Then order them by PageRank (from highest to lowest).

Solution. We have to follow the four steps described in exercise *A 4-page network*, page 112. For that we need to call up in Octave the instructions listed in Script 4.3, page 117, but replace A by the matrix associated with the graph of this exercise, namely the matrix A obtained in Exercise 4.1, page 216.

1. We define in Octave the matrix A of the graph

```
A=[ 0 0 0 1 1 0;
    1 0 1 1 1 0;
    0 1 0 1 0 0;
    1 0 0 0 1 0;
    0 1 0 0 0 0;
    0 1 1 0 1 0]
```

2. We build the corresponding Google matrix with $\alpha = 0.85$:

```
G=matrix_G(A,0.85)
```

3. We compute the PageRank by: calling the `compute_pagerank` function (which implements Algorithm 4 on page 111), and fixing the tolerance $\epsilon = 10^{-3}$ and the maximum number of iterations $k_{max} = 100$:

```
[p,k]=compute_pagerank(G,1e-3,100)
```

The method converges after `k=9` iterations, and the PageRank vector is

```
p =
   0.12034
   0.24037
   0.16203
   0.13362
```

```
    0.12417
    0.21946
```

4. Finally, we reorder the components of `p` by the command

```
[psort,ranking]=sort(p,'descend')
```

to get

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

Therefore: the page with highest PageRank is page number 2, then come pages 6, 3, 4, 5 and at last page 1.

□

Exercise 4.4. Consider the matrix A of Exercise 4.2 and the associated graph. Compute the PageRank of the pages of the network modelled by such graph. Then order the pages in terms of PageRank (from highest to lowest).

Solution. We have to follow the four steps described in exercise *A 4-page network* and hence call up in Octave the instructions of Script 4.3, where A is replaced by the matrix associated with the graph of the present exercise, i.e. the matrix A assigned in Exercise 4.2, page 217.

1. Define in Octave the matrix A of the graph

```
A=[ 0 1 1 1 0 0 0;
    0 0 1 0 0 0 1;
    1 1 0 1 0 1 0;
    1 0 1 0 0 1 0;
    0 1 0 1 0 0 1;
    0 0 0 0 1 0 0;
    0 1 0 0 0 0 0]
```

2. Build the corresponding Google matrix with $\alpha = 0.85$:

```
G=matrix_G(A,0.85)
```

3. Compute PageRanks by: calling up the `compute_pagerank` function (which implements Algorithm 4, page 111), and fixing the tolerance $\epsilon = 10^{-3}$ and the maximum number of iterations $k_{max} = 100$

```
[p,k]=compute_pagerank(G,1e-3,100)
```

The method converges after `k=9` iterations, and the PageRank vector reads

```
p =
    0.167649
    0.104417
    0.227514
    0.210206
    0.121725
    0.124865
    0.043623
```

4. Finally we reorder the components of `p` with the command

```
[psort,ranking]=sort(p,'descend')
```

and eventually get

```
ranking =
    3
    4
    1
    6
    5
    2
    7
```

The page with the highest PageRank is page number 3, followed by pages 4, 1, 6, 5, 2 and 7.

□

8.4 Exercises of Chapter 5

Exercise 5.1. Determine the average speed $\overline{v}$ of blood in a capillary modelled by a rigid cylinder of length $L = 0.5$ mm and radius $r = 3\,\mu$m, knowing that the viscosity is 2 cP and the blood pressure at the endpoints is 40 mmHg and 38 mmHg.

The centipoise (cP) is the unit of measure of viscosity (a fraction of the poise): 1 cP = 10^{-3}Pa s = $10^{-3}\dfrac{\text{g}}{\text{mm s}}$. We have already seen that the millimetre of mercury (mmHg) is the unit of measure of pressure: 1 mmHg = $133.32\dfrac{\text{g}}{\text{mm s}^2}$.

Solution. We use mathematical model (5.3)

$$\overline{v} = \frac{(p_1 - p_2)\, r^2}{8\mu\, L}.$$

Substituting the data of the current problem, we find

$$\overline{v} = \frac{(40 - 38)\text{ mmHg } \cdot 9\ (\mu\text{m})^2}{8 \cdot 2\text{ cP } \cdot 0.5\text{ mm}}.$$

Before we proceed with the calculations, let us express millimetres of mercury (mmHg) and centipoise (cP) in terms of millimetres, grams and seconds only.

Recalling that 1 mmHg $= 133.32\frac{\text{g}}{\text{mm s}^2}$, 1 µm $= 10^{-3}$ mm and 1 cP $= 10^{-3}\frac{\text{g}}{\text{mm s}}$, we have

$$\overline{v} = \frac{(40-38)\text{ mmHg } \cdot 9\ (\mu\text{m})^2}{8 \cdot 2\text{ cP } \cdot 0.5\text{ mm}} = \frac{2 \cdot 133.32\frac{\text{g}}{\text{mm s}^2} \cdot 9 \cdot 10^{-6}\text{ mm}^2}{8 \cdot 2 \cdot 10^{-3}\frac{\text{g}}{\text{mm s}} \cdot 0.5\text{ mm}}$$
$$= \frac{133.32 \cdot 9 \cdot 10^{-3}}{4}\frac{\text{mm}}{\text{s}} = 0.3\frac{\text{mm}}{\text{s}}.$$

Hence the average speed inside the capillary is 0.3 mm/s. □

Exercise 5.2. Consider the data of Exercise 5.1 and compute the resistance R to the motion of the fluid. Determine the resistance R and the average speed of blood when the radius of the capillary is reduced by a tenth, from $r = 3$ µm to $r = 2.7$ µm. How does the average speed vary in terms of the radius? What is the behaviour of the resistance as the capillary radius varies?

Solution. First of all we find the resistance R using the data of Exercise 5.1, that is: radius $r = 3$ µm and viscosity $\mu = 2\text{ cP} = 2 \cdot 10^{-3}\frac{\text{g}}{\text{mm s}}$. So,

$$R = \frac{8\mu}{\pi r^4} = \frac{8 \cdot 2 \cdot 10^{-3}\frac{\text{g}}{\text{mm s}}}{\pi\ (3\ \mu\text{m})^4} = 6.2876 \cdot 10^7\frac{\text{g}}{\text{mm}^5\text{ s}}.$$

Now we compute the resistance and average speed with $r = 2.7$ µm, $L = 0.5$ mm, $\mu = 2\text{ cP} = 2 \cdot 10^{-3}\frac{\text{g}}{\text{mm s}}$ and $(p_1 - p_2) = 2\text{ mmHg} = 2 \cdot 133.32\frac{\text{g}}{\text{mm s}^2}$. To distinguish these from the values of the previous exercise we shall denote the former by R_2 and $\overline{v}_2$:

$$R_2 = \frac{8\mu}{\pi r^4} = \frac{8 \cdot 2 \cdot 10^{-3}\frac{\text{g}}{\text{mm s}}}{\pi\ (2.7\ \mu\text{m})^4} = 9.5833 \cdot 10^7\frac{\text{g}}{\text{mm}^5\text{s}},$$
$$\overline{v}_2 = \frac{(40-38)\text{ mmHg } (2.7\ \mu\text{m})^2}{8 \cdot 2\text{ cP } 0.5\text{ mm}} = \frac{2 \cdot 133.32\frac{\text{g}}{\text{mm s}^2} \cdot 7.29 \cdot 10^{-6}\text{mm}^2}{8 \cdot 2 \cdot 10^{-3}\frac{\text{g}}{\text{mm s}}\ 0.5\text{mm}}$$
$$= \frac{133.32 \cdot 7.29 \cdot 10^{-3}}{4}\frac{\text{mm}}{\text{s}} = 0.24298\frac{\text{mm}}{\text{s}}.$$

As the radius decreases the resistance increases, whilst the speed has decreased. This agrees with the definition of resistance formulated on page 129 (whereby the resistance is proportional to the negative fourth power of the radius) and with formula (5.3) (which expresses the fact that the average speed in the capillary is proportional to the radius r squared). □

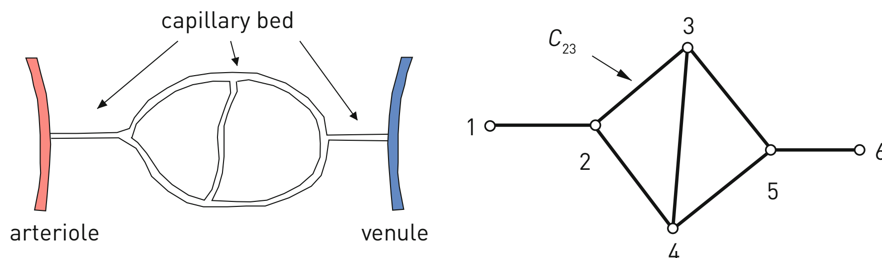


Fig. 8.2 On the left, an example of a simple capillary bed. On the right, the corresponding graph of capillaries and nodes: nodes 2 and 3 are the endpoints of capillary C_{23}

Exercise 5.3. Consider the capillary bed of Fig. 8.2, whose capillaries all have radius $r = 4$µm and lengths:

Capillary C_{ij}	C_{12}	C_{23}	C_{24}	C_{34}	C_{35}	C_{45}	C_{56}
L_{ij} (mm)	0.3	0.3	0.4	0.5	0.4	0.3	0.3

Assume the viscosity of blood equals 2 cP and the pressure at nodes 1 (at the arteriole) and 6 (at the venule) are respectively $p_a = 40$ mmHg and $p_v = 15$ mmHg. Compute the pressures at the nodes of the capillary bed, the volumetric flow rate and the average speed along each capillary.

Solution. The steps to follow are those indicated in exercise *A 7-capillary bed* on page 139, namely:

1. define the data of the problem and build the matrix G of the graph;
2. compute the pressures at the nodes of the graph by solving the linear system $A\mathbf{p} = \mathbf{b}$ (calling up the `pressures` function);
3. compute the volumetric flow rate Q in each capillary using formula (5.6), page 139 (calling up the `flow_rates` function);
4. compute the average speed $\bar{v}_{ij}$ in each capillary by formula (5.7), page 139 (calling up the `average_speeds` function).

To define the data of the problem we write an Octave script that is very much like script `data_7bed` (see Script 5.4, page 150), which was prepared for exercise *A 7-capillary bed*.

To that end, in the menu on the lower part of the Octave window we click on "Editor". In the window that opens we then type the following instructions:

```
r=4e-3; % capillary radius (mm)
mu=2e-3; % blood viscosity g/(mm*s)
pa=40; pv=15; % pressure in the arteriole and venule (mmHg)
R=8*mu/(pi*r^4); % resistance in the capillaries g/(mm^5*s)
Area=pi*r^2; % area of the capillary cross-section (mm^2)
% G = graph matrix with the lengths of the capillaries
G=[0   0.3 0   0   0   0;
   0.3 0   0.3 0.4 0   0;
```

```
    0    0.3 0    0.5 0.4 0;
    0    0.4 0.5 0    0.3 0;
    0    0    0.4 0.3 0    0.3;
    0    0    0    0    0.3 0];
disp('The data of the exercise have been defined')
```

The symbol % ensures that everything that comes after it up to the end of the line is considered a comment, and hence is not interpreted. To better understand the Octave instructions we refer to Sect. 3.1, Chap. 3.

Select from the menu "File > Save File As", then save in a script (for instance called data_bed_bis.m) the typed instructions. Make sure the file name has the extension .m, the typical ending of an Octave file.

In the menu on the lower part of the Octave window, click on "Command Window" to return to the Octave command window, and type the command

```
data_bed_bis
```

If we did not make mistakes when writing the instructions, Octave should reply with the string

```
The data of the exercise have been defined
```

If, instead, an error message appears, we must go back to the Editor window and amend the possible mistakes, save, return to the "Command Window" and execute the command data_bed_bis again.

Once the data have been built, we compute the pressures at the nodes of the bed by invoking the function pressures by the instruction

```
p=pressures(G,pa,pv);
```

Octave displays:

```
The pressure at node 1 equals  40.0000 mmHg
The pressure at node 2 equals  32.0807 mmHg
The pressure at node 3 equals  27.8882 mmHg
The pressure at node 4 equals  27.1118 mmHg
The pressure at node 5 equals  22.9193 mmHg
The pressure at node 6 equals  15.0000 mmHg
```

Next we compute the volumetric flow rates of the capillaries with the instruction

```
Q=flow_rates(G,p,R);
```

and Octave reacts as follows:

```
The volumetric flow rate in C_12 is 1.769e-04 mm^3/s
The volumetric flow rate in C_23 is 9.365e-05 mm^3/s
The volumetric flow rate in C_24 is 8.325e-05 mm^3/s
The volumetric flow rate in C_34 is 1.041e-05 mm^3/s
The volumetric flow rate in C_35 is 8.325e-05 mm^3/s
The volumetric flow rate in C_45 is 9.365e-05 mm^3/s
The volumetric flow rate in C_56 is 1.769e-04 mm^3/s
```

Finally we compute the speeds with the instruction

```
v=average_speeds(G,Q,Area);
```

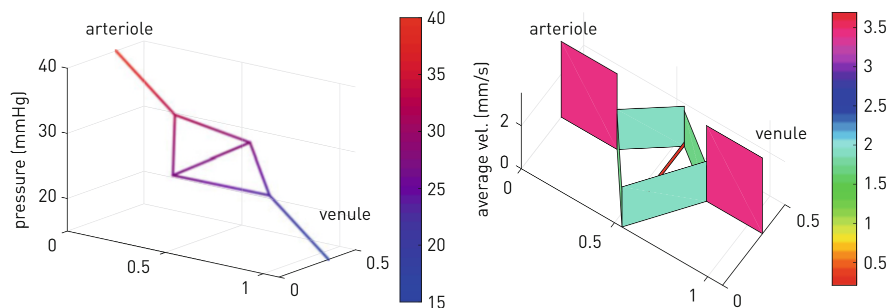


Fig. 8.3 The solutions of exercise 5.3. Left: the pressures in the capillary bed in mmHg (nodal values have been joined by segments). Right: the average speeds in mm/s (constant inside each capillary)

and get

```
The average speed in C_12 is 3.5193 mm/s
The average speed in C_23 is 1.8632 mm/s
The average speed in C_24 is 1.6561 mm/s
The average speed in C_34 is 0.2070 mm/s
The average speed in C_35 is 1.6561 mm/s
The average speed in C_45 is 1.8632 mm/s
The average speed in C_56 is 3.5193 mm/s
```

To plot pressures and average speeds we type

```
plot_7bed(G,p,v);
```

The plots are shown in Fig. 8.3. Since it is present in the folder of Octave files of the book, we did not reproduce the `plot_7bed` function here. □

Exercise 5.4. Consider a "grid-like" capillary bed, similar to that of Fig. 5.17 on page 143. Instead of having 4×4 capillaries (plus the first and last, which are connected to the arteriole and venule respectively), this one is made of 8×8 capillaries (plus the first and last, joint to the arteriole and venule). Suppose the capillaries have length $L = 0.5$ mm and radius $r = 3$ µm, the viscosity is $\mu = 2$ cP, the pressures at the arteriole and venule $p_a = 40$ mmHg and $p_v = 15$ mmHg respectively. Determine the pressures at the nodes of the network and the average speeds in the capillaries, then plot them.

Solution. For this exercise we can use script `data_grid_like_bed` (see Script 5.5 at page 151), which has been prepared to build the matrix of the graph associated with a "grid-like" capillary bed with an arbitrary number $(n_c \times n_c) + 2$ of capillaries (the last two are those inserting on the arteriole and venule).

From the menu on the lower part of the Octave window we click on "Editor" to open a new window, then in the next menu on the top we click on the icon to open an existing file (or select "File > Open"). Then we select the `data_grid_like_bed.m` file from the list. The Editor window displays the

content of data_grid_like_bed.m, which is exactly what is printed in Script 5.5 on page 151. Here we can modify the data and add new instructions.

In particular, we can modify the radius r and the viscosity μ by replacing the lines

```
r=2.5e-3; % radius of the capillaries (mm)
mu=1.9e-3; % blood viscosity g/(mm*s)
```

by

```
r=3e-3; % radius of the capillaries (mm)
mu=2e-3; % blood viscosity g/(mm*s)
```

We save the file under a new name (for example, data_grid_like_bed_bis.m) by selecting "File > Save File As" in the menu (make sure the file name ends with .m, the extension of Octave documents). Now, from the menu on the lower part of the Octave window we click on "Command Window" to return to the command window of Octave.

Since the capillary bed under consideration is made by $(8 \times 8) + 2$ capillaries, we must initialise the variable nc with the value 8 before we call the data_grid_like_bed_bis script.

The instructions are:

```
nc=8; data_grid_like_bed_bis;
```

At this point we may compute the pressures (calling up the pressures function), the volumetric flow rates (with flow_rates), the average speeds (with average_speeds), then plot pressures and speeds using the plot_grid_like_bed function.
The Octave instructions are:

```
p=pressures(G,pa,pv);
Q=flow_rates(G,p,R);
v=average_speeds(G,Q,Area);
plot_grid_like_bed(G,p,v)
```

and the output is shown in the following figure.

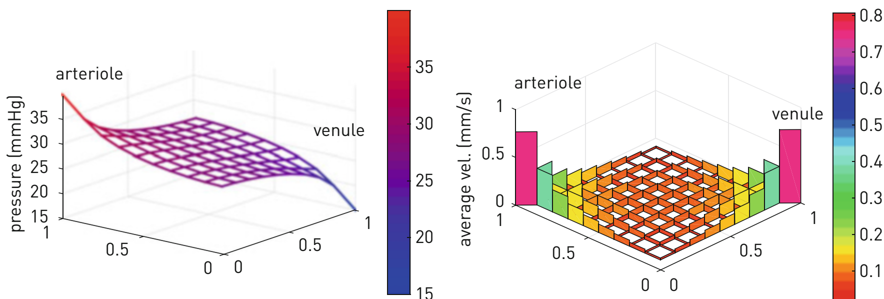


Fig. 8.4 The solutions to exercise 5.4. On the left, the pressures in the capillary bed in mmHg (nodal values were joint by segments). On the right, the average speeds in mm/s (a constant value inside each capillary)

□

8.5 Exercises of Chapter 6

Exercise 6.1. The table below shows the demographic evolution in Togo (a central African state) from 1960 to 2016:[1]

Year	Population
1960	1,580,513
1970	2,115,522
1980	2,720,839
1990	3,786,940
2000	4,970,367
2010	6,502,952
2016	7,606,374

Verify that in the 1960–2016 frame the population has grown according to exponential model (6.2), page 160, with growth rate $r = 0.0283$. Predict what the population will be in 2025.

Solution. Let us consider exponential model (6.2) with the data at our disposal: $t_0 = 1960$, $y_0 = 1,580,513$ and $r = 0.0283$. We must verify the data of the table are described (up to small errors) by the function

$$y(t) = y_0\, e^{r(t-t_0)} = 1,580,513\, e^{0.0283(t-1960)}.$$

We compute the values of $y(t)$ (and round them off to the closest integer) in correspondence to the years in the table, to get the values reported in Table 6.1. By observing the plot of Fig. 6.5, the actual measurements agree very well with the values $y(t)$ found by the exponential model (see also Table 6.1). However, in

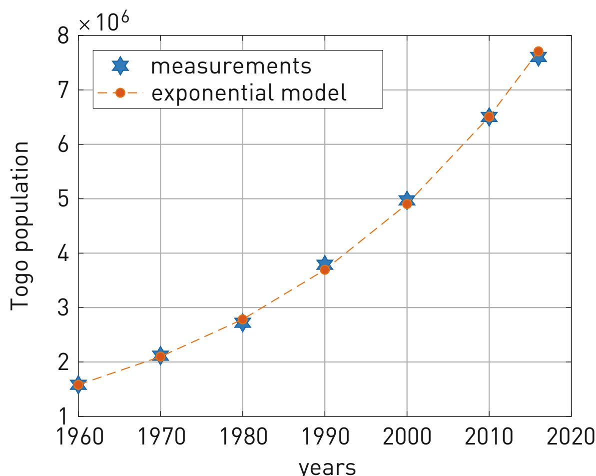


Fig. 6.5 Actual measurements and numerical results of Exercise 6.1

[1] Source: World Development Indicators, https://data.worldbank.org

Table 6.1 Numerical results of Exercise 6.1

t	$y(t)$
1960	1,580,513
1970	2,097,507
1980	2,783,612
1990	3,694,146
2000	4,902,521
2010	6,506,160
2016	7,710,241

order for this result to be quantitatively (and not just qualitatively) significant, for every year t for which data are available, we compute the relative error between the measurement (which we shall call $population(t)$) and the value $y(t)$ provided by the model. In other words we calculate the quantity:

$$e(t) = \frac{|y(t) - population(t)|}{population(t)}, \quad \text{for } t = 1960, \dots, 2016$$

Hence:

Year t	1960	1970	1980	1990	2000	2010	2016
Error $e(t)$	0	0.0085	0.0231	0.0245	0.0137	0.0005	0.0137

The maximum error, 2.45%, should be considered satisfactory.

Finally, we evaluate $y(t)$ at $t = 2025$ and obtain $y(2025) = 9,946,786$. Therefore in 2025 the population of Togo is expected to reach around 10 millions. □

Exercise 6.2. As it happens, dolphins are unfortunately trapped in fishing nets, killed by speedboat blades, or die as a consequence of sea pollution. Let us consider the data of exercise *Aeolian dolphins* on page 161 ($t_0 = 2017.5$, $y_0 = 100$ and $r = 0.06$), and denote by B the number of individuals that fall victim to these incidents every year. Determine the largest-possible value of B that ensures the survival of the dolphin population in the Aeolian sea.

Solution. To describe the evolution of the dolphin population when a part of it (expressed by the number B) falls victim to incidents, we shall use model (6.3) on page 163.

The survival of the species is guaranteed if the solution $y(t)$ to model (6.3) is not a decreasing exponential function, i.e. if $B \leq r\, y_0$. Using the data of our problem we obtain

$$B \leq r\, y_0 = 0.06 \cdot 100 = 6,$$

so the dolphin population will survive provided no more than 6 dolphins per year perish to accidents caused by mankind. □

Exercise 6.3. Compute the numerical solution to the Cauchy problem of page 161 for different values of h. Compare it with the exact solution provided, and check that the Euler method converges with order 1 with respect to h.

Solution. We have prepared script `exercise_dolphins1`, in which we define the data of the Cauchy problem on page 161. Moreover, for each $h \in \{0.1, 0.05, 0.01, 0.005, 0.001\}$ we compute $N_h = \dfrac{t_f - t_0}{h}$, call up the `euler` function and calculate the error

$$e(h) = \max_{0 \le n \le N_h} |u_n - y(t_n)|. \tag{6.5}$$

Typing in Octave the command

```
exercise_dolphins1
```

gives Fig. 8.6.

Since the error line (blue) is parallel to the line $y = h$ (red), we conclude that the error of the Euler method behaves like h, as h tends to 0. We reach the same conclusion by observing that as we lower the order of magnitude of h by one, the error, too, decreases by one order of magnitude. Therefore we deduce that the Euler method converges with respect to h with order $p = 1$.

Let us now make a few comments concerning the Octave commands used in Script 8.2, page 229, which have not yet been discussed.

To define the function $f(t, y)$ we have used a function handle just like we did for functions of one variable only. But now f depends on two variables, t and also y.

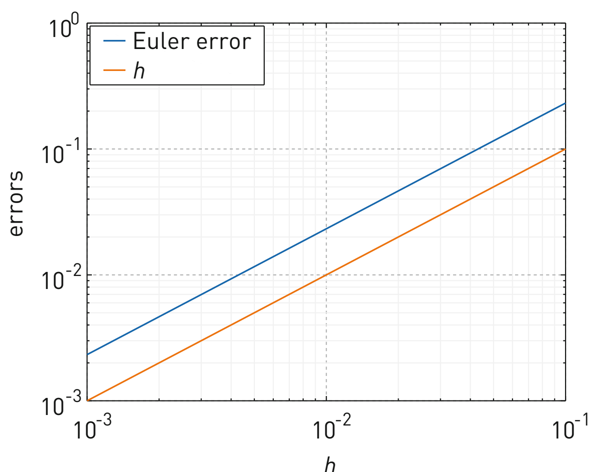


Fig. 8.6 The output of Exercise 6.3

Note that y does not depend on t but plays the role of an independent variable, and both variables must be specified in the block `@(t,y)`, even if the expression will not explicitly depend on t.

Beside the function handle `f`, we also have defined the function handle `yex`, associated with the exact solution of the Cauchy problem.

To define the value of `Nh` we have used the `fix` command, which rounds off to the closest integer the value of its argument. Because of the finite arithmetic of the computer, in fact, nothing guarantees that the result of the operation $\frac{t_f - t_0}{h}$ will be an integer. □

`fix`

Exercise 6.4. Compute the numerical solution of the Cauchy problem on page 161 using the Euler method with $h = \frac{1}{12}$. Plot said solution and compare it with the one of the Cauchy problem of page 176, which was also obtained with $h = \frac{1}{12}$.

Solution. The Octave instructions to solve the exercise are gathered in script `exercise_dolphins2`. In it we defined the endpoints of the time interval and the initial condition, and we fixed the discretisation step $h = \frac{1}{12}$. Then we defined the function $f(t, y) = 0.06y$ for the Cauchy problem of page 161, and

```
t0=2017.5; tf=2025.5; % endpoints of the time interval
y0=100;         % initial condition
f=@(t,y)0.06*y; % function f(t,y)
yex=@(t)y0*exp(0.06*(t-t0)); % exact solution
H=[0.1, 0.05, 0.01, 0.005, 0.001]; % vector of time steps
Err=[ ];
for h=H
Nh=fix((tf-t0)/h); % computes Nh
% compute the numerical solution by the Euler method
[tn,un]=euler(f,t0,tf,y0,Nh);
yn=yex(tn); % evaluates the exact solution at the grid nodes
err=max(abs(un-yn)); % computes the error for the given h
Err=[Err, err]; % appends err at the end of vector Err
end
% plot the behaviour of errors
figure(1); clf % selects and wipes the graphical window
loglog(H,Err,'Linewidth',3); % plots errors in function of h
grid on; hold on; % plots the grid and keeps the graph
loglog(H,H,'Linewidth',3); % plots the line y=h
xlabel('h','Fontsize',18); ylabel('errors','Fontsize',18);
le=legend('Euler error','h'); % defines the legend
set(le,'Fontsize',18,'Location','Northwest')
set(gca,'Fontsize',18)
```

.

Script 8.2: `exercise_dolphins1.m`: script to solve Exercise 6.3

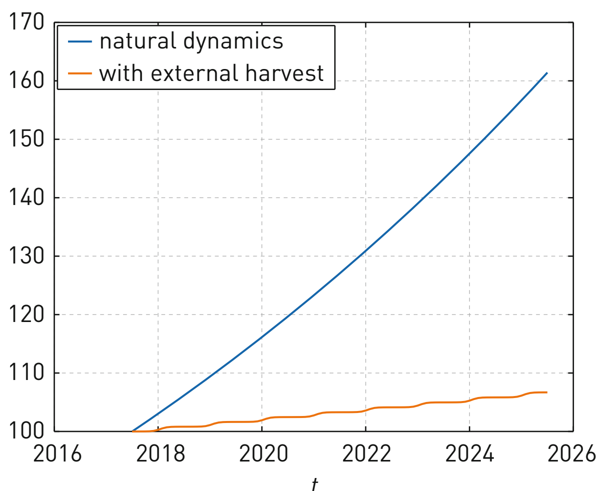


Fig. 8.7 The output of Exercise 6.4. In blue the solution to the Cauchy problem of page 161, in red the solution to the Cauchy problem of page 176.

computed the numerical solution by the Euler method. Next, we defined the function $f(t, y) = 0.06y + b(t)y$ for the Cauchy problem of page 176 with $b(t)$ as indicated in exercise *Aeolian dolphins (2): the effect of the environment* (page 176). Finally we computed the numerical solution with the Euler method. Figure 8.7 shows

```
t0=2017.5; tf=2025.5; % endpoints of the time interval
y0=100; % initial condition
h=1/12;  % define the step h
Nh=fix((tf-t0)/h); % computes Nh
% solve the problem for the natural dynamics
f1=@(t,y)0.06*y; % function f(t,y) of the natural dynamics
% compute the numerical solution with the Euler method
[tn1,un1]=euler(f1,t0,tf,y0,Nh);
% solve the problem for the natural dynamics with external
% harvest
b=@(t)-0.03*(2-exp(-5*(sin(t*pi))^2)); % defines b(t)
% function f(t,y) for the natural dynamics with external harvest
f2=@(t,y)(0.06+b(t))*y;
% compute the numerical solution with the Euler method
[tn2,un2]=euler(f2,t0,tf,y0,Nh);
% plot the solution
figure(1); clf % selects and wipes the graphical window
plot(tn1,un1,'Linewidth',2)
grid on; hold on; % plots the grid and keep the graph
plot(tn2,un2,'Linewidth',2)
xlabel('t','Fontsize',16);
le=legend('natural dynamics',...
          'with external harvest'); % defines the legend
set(le,'Fontsize',16,'Location','Northwest')
set(gca,'Fontsize',16)
```

Script 8.3: `exercise_dolphins2.m`: script for solving exercise 6.4.

the numerical solutions thus obtained. The external harvest manifestly stymies the growth of the dolphin population. □

Exercise 6.5. Suppose now that the function $b(t)$ introduced in exercise *Aeolian dolphins (2): the effect of the environment*, page 176, is replaced by

$$b(t) = -0.06(2 - e^{-5(\sin(t\pi))^2}).$$

Once again we wish to estimate the number of dolphins in July 2025, assuming that in July 2017 there are 100 exemplars and the growth rate stays $r = 0.06$. Choose h so to guarantee the solution is sufficiently accurate.
How is the dolphin population evolving?

Solution. Script `exercise_dolphins3` contains the Octave instructions for this exercise. In this script we defined the endpoints of the time interval, the initial condition and also fixed the time step $h = \frac{1}{12}$. Then we defined the function $f(t, y) = 0.06y + b(t)y$, where $b(t)$ is as indicated in the text, and computed the numerical solution using the Euler method. The numerical solution is plotted in Fig. 8.8, which clearly shows that in this simulation dolphins will become extinct. □

Exercise 6.6. Find the computational solution to exercise *Hakes and sharks (2): fishing* on page 189, with $b_d \in \{0.4, 0.5, 0.6\}$. Plot the numerical solutions thus obtained and verify the conclusions of Table 6.2, page 191.

Solution. To solve this exercise it suffices to modify few instructions in function `hakes_sharks_fishing` (see Function 6.7, page 198).

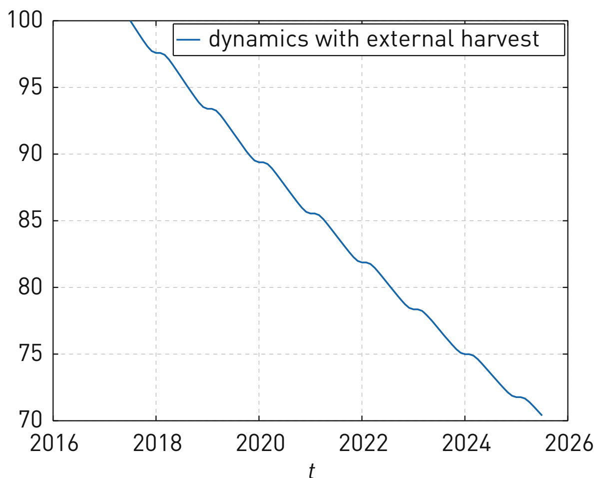


Fig. 8.8 The output of exercise 6.5

```
t0=2017.5; tf=2025.5; % endpoints of the time interval
y0=100; % initial condition
h=1/12;  % defines the step h
Nh=fix((tf-t0)/h); % computes Nh
% solve the problem for the natural dynamics with external
% harvest
b=@(t)-0.06*(2-exp(-5*(sin(t*pi))^2)); % defines b(t)
% function f(t,y) of the natural dynamics with external harvest
f=@(t,y)(0.06+b(t))*y;
% compute the numerical solution using the Euler method
[tn,un]=euler(f,t0,tf,y0,Nh);
% plot the solution
figure(1); clf % selects and wipes the graphical window
plot(tn,un,'Linewidth',2); grid on
xlabel('t','Fontsize',16);
le=legend('dynamics with external harvest'); % defines the
%   legend
set(le,'Fontsize',16,'Location','Northeast')
set(gca,'Fontsize',16)
```

Script 8.4: `exercise_dolphins3.m`: script for Exercise 6.5

In the menu on the lower part of the Octave window we click on "Editor", and in the new menu that opens we click on the icon to open an existing file (or select "File > Open"). Then we select the file `hakes_sharks_fishing.m` from the list. The Editor window now displays the contents of file `hakes_sharks_fishing.m`, and operating in the Editor window allows us to modify and/or add new instructions.

First things first, the instruction

```
for bd=[0.3,0.1,0]
```

should be replaced by:

```
for bd=[0.4,0.5,0.6]
```

This changes the choice of the coefficient b_d responsible for the fishing. Consequently, we also have to modify the instruction specifying the legend strings, i.e.

```
legend('bd=0.3','bd=0.1','bd=0')
```

should be replaced (twice) by

```
legend('bd=0.4','bd=0.5','bd=0.6')
```

Then we save the new function, for instance under the name

`hakes_sharks_fishing_bis.m` (as a matter of fact, to simplify your work we have prepared such a function already, see Function 8.1 at the end of the exercise. You may pick a name of your choosing).

To save the modified file, select "File > Save File As" from the menu (making sure the file name ends with `.m`, the extension of Octave documents). At this point, in the menu on the lower part of the Octave window we click on "Command Window" to go back to the Octave command window.

Fixing $N_h = 1000$ and typing in Octave the command

```
hakes_sharks_fishing_bis(1000)
```

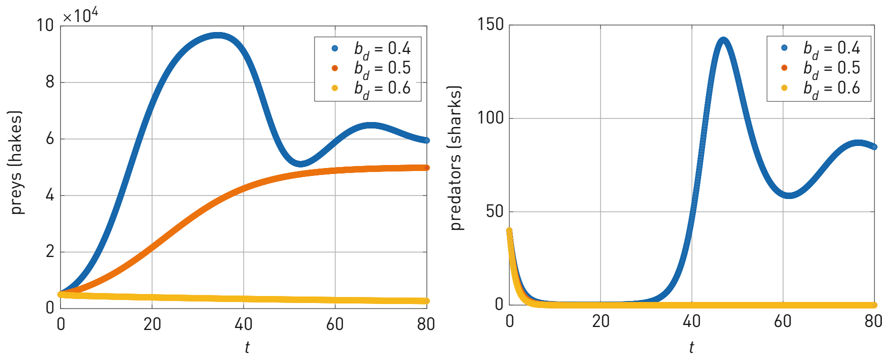


Fig. 8.9 The output of exercise 6.6

```
function hakes_sharks_fishing_bis(Nh)
% hakes_sharks_fishing_bis: solves problem "Hakes and Sharks
%  with harvest"
% Calling instruction: hakes_sharks_fishing_bis(Nh)
% Input: Nh = number of time steps
% Output: graphic window

% define the problem data
t0=0; tf=80; % endpoints of the time interval
p0=5000; s0=40; % initial conditions
rp=.6; rs=-.2; Kp=3.e5; sigmap=rp/Kp; % data
sigmas=1.e-4; alpha=0.001; mu=.01;
figure(1); clf; figure(2); clf; % opens and wipes figures
for bd=[0.4,0.5,0.6] % loop of three different values of bd
f1=@(t,p,s)(rp-sigmap*p-alpha*s-bd)*p;
f2=@(t,p,s)(rs-sigmas*s+mu*alpha*p-bd)*s;
% compute the numerical solution by the Euler method
[tn,un,vn]=euler2(f1,f2,t0,tf,p0,s0,Nh);
% plot the numerical solution
figure(1);
plot(tn,un,'Linewidth',2);
hold on; grid on
figure(2);
plot(tn,vn,'Linewidth',2);
hold on; grid on
end
figure(1);
xlabel('t'); ylabel('preys (hakes)');
legend('bd=0.4','bd=0.5','bd=0.6')
figure(2);
xlabel('t'); ylabel('predators (sharks)');
legend('bd=0.4','bd=0.5','bd=0.6')
```

Function 8.1: `hakes_sharks_fishing_bis.m`: function for exercise 6.6

produces the output of Fig. 8.9, and we can verify the conclusions written in Table 6.2, page 191. □

Index

Symbols
', 62
.*, 67
..., 213
./, 67
.^, 67
;, 56
», 54
[ ], 77
\, 138
%, 56
^ * / -, 55

A
A(2,:), 61
A(:,3), 62
a:step:b, 64, 75
A=[ ], 60
abs, 66
addpath, 71
Algorithm, 39, 44
  Cramer, 38
  for loop, 39, 40
  GEM, 44, 49
  matrix of a graph, 137, 145
  PageRank, 100, 111
  variable assignment, 40
  while loop, 111
ans, 56
Approximation, 4, 8, 11, 13, 18, 20, 109, 184
  of a differential equation, 171
  of the first derivative, 165, 175
Arithmetic
  exact, 10, 92
  floating-point, 10, 92
Array, 59
Assignment of variable, 60
Average speed, 129
axis, 86

B
Backslash, 138
Base, 88
Blood pressure, 123
Browser, 104

C
Carrying capacity of the environment, 179
Comment line, 56
Complexity, 2
Component of a vector, 27, 38
Computational cost, 48
Computer precision, 11, 82, 83, 90
Convergence, 12, 168
Convergence order, 167

D
Data
  of a mathematical model, 4
  of a physical problem, 4, 5
Derivative, 165
det, 213
Determinant, 31, 37
Dimension
  of a matrix, 59
  of a vector, 28
Discretisation
  parameter, 11, 12, 22
  step, 172
disp, 70
Dynamics
  natural, 160
  of a population, 165

E
Eigenvalues, 108
Element
  diagonal, 29, 136
  of a matrix, 29
  off-diagonal, 29, 136

© Springer Nature Switzerland AG 2020

A. Quarteroni, P. Gervasio, *A Primer on Mathematical Modelling*, UNITEXT 121,
https://doi.org/10.1007/978-3-030-44541-6

Equations
- balance, 134
- differential, 159
- Lotka-Volterra, 184
- Navier-Stokes, 4
- Poisson, 6

Error, 175
- absolute, 91
- computational, 10
- estimator, 14, 23, 110
- of the Euler method, 175
- of the finite difference, 167
- of the model, 8, 11
- numerical, 8, 11–13, 22, 174
- of quadrature formula, 23
- relative, 91
- roundoff, 10, 11, 90

`exist`, 78
`exp`, 58
Exponential form of a number, 88
Exponent of a floating point, 88

F

Factorial, 46, 80
`fix`, 229
Floating point, 88
Fluid
- dynamics, 4
- ideal, 129
- Newtonian, 126

`for`, 75
For loop, 40, 44, 75
`format`, 55
Forward finite difference, 166
- error, 167

`fplot`, 58
`fprintf`, 70
Function
- differentiable, 166
- handle, 58, 228
- mathematical in Octave, 58
- of Octave, 72
- step, 19

G

Gauss elimination method (GEM), 32, 41, 44, 49, 138, 215
`gca`, 84
Graph, 131
- directed, 101, 131
- undirected, 131

H

`help`, 59

I

`if...end`, 77
`if...else...end`, 78
`if...elseif...else...end`, 78
Infinitesimal calculus, 13, 22
Initial condition, 160
Iteration, 109
Iterative process, 109, 112

K

Keyword, 56, 79

L

Law
- exponential evolution, 159
- Malthus, 159
- Poiseuille, 128
- total probability, 106

`legend`, 59
Limit, 12
Linear system, 31, 45, 135, 136
- back-substitution, 33
- determined, 44
- matrix form, 31
- order, 38
- reduction, 33

`linspace`, 68
`loglog`, 85
Loop
- body of a, 112
- for, 39, 40, 44, 75
- while, 81, 111, 112

M

Machine epsilon, 90
Mantissa of a floating point, 88
Mathematical modelling, 2
Matrix, 28, 29, 59
- adjacency, 102, 131
- column of a, 29
- column-stochastic, 105
- dimension of a, 59
- element of a, 29
- form of a linear system, 31
- Google, 105
- of a graph, 102, 131
- Hilbert, 76
- main diagonal, 29
- order of a, 29
- rectangular, 59
- row of a, 29
- sparse, 115
- square, 29, 59
- symmetric, 132

transpose, 62
upper triangular, 210
max, 65
Method
convergent, 12, 168, 175
convergent of order p, 13
Cramer, 31, 36, 38
Euler, 173
error, 175
for systems, 184, 196
Gauss elimination (GEM), 32, 41, 138, 215
numerical, 2
of order p, 170
substitution, 26
min, 65
Model, 6
Lotka-Volterra, 184
mathematical, 2, 4, 16, 31, 36, 45, 108, 136, 165, 180, 184, 188
of natural dynamics, 160
numerical, 2, 17, 23, 112
verification of, 10
Multiplier, 33

N
Name of a variable, 56
Number
exponential form of a, 83, 88
floating point, 88

O
Octave statement, 75, 77, 81
Operations
element-by-element, 67
elementary, 22, 45, 46
in Octave, 55
Operators
logical, 81
short-circuit, 81
relational, 79
Order
of convergence, 13, 168, 170
of a linear system, 26, 38
of a matrix, 29
Overflow, 90

P
PageRank, 99
algorithm, 100
Paraboloid, 128
Path, 71
Pivoting, 44
plot, 68
plotyy, 85
Probability, 100
conditional, 106
Problem
Cauchy, 160
numerical, 23
physical, 2
prod, 65
Product
matrix-vector, 30, 34, 63
Program, 47
pt (typographic point), 84

Q
Quadrature
of the circle, 18
nodes, 18, 21
Quadrature formula, 17, 18
midpoint, 20
error, 23

R
Rate
birth, 158
death, 158
depletion, 165, 188
growth, 158
predation, 183
prey-to-predator conversion, 184
Resistance, 129
Results
of a mathematical model, 4
of a physical problem, 5
return, 214
Roundoff unit, 11, 91

S
Scale
linear, 169, 171
logarithmic, 85, 94, 169, 170
semi-logarithmic, 94
Scientific computing, 3
Script, 71
Selection blocks, 77, 138, 212
semilogy, 94
Sequence, 93
set, 84
Short-circuit, 81
size, 212
Solution, 4
computational, 11, 12
existence and uniqueness, 4
of the mathematical model, 2
numerical, 2, 12, 20, 110, 185
of a physical problem, 3, 4

sqrt, 58
Standard deviation, 16
Stopping test, 110
String, 59
Sum, 20
  of matrices, 62
sum, 65
System
  linear, 26, 31, 35
  order, 26
  prey-predator, 184, 188

T
Tends to..., 12
Transpose, 62

U
Underflow, 90
User-defined function, 72

V
Validation of the process, 10
Variable, 40, 55
  assignment of a, 40
  of logical/Boolean type, 79
Vector, 30
  column, 27, 59
  component of a, 27, 38
  dimension of a, 28
  right-hand-side, 27
  row, 28, 59
  solution, 27
Verification of the numerical model, 10, 169
Viscosity, 126
Volumetric flow rate, 128

W
warning, 214
while, 81
While loop, 81, 111

X
xlabel, 59

Y
ylabel, 59

Z
zeros, 76