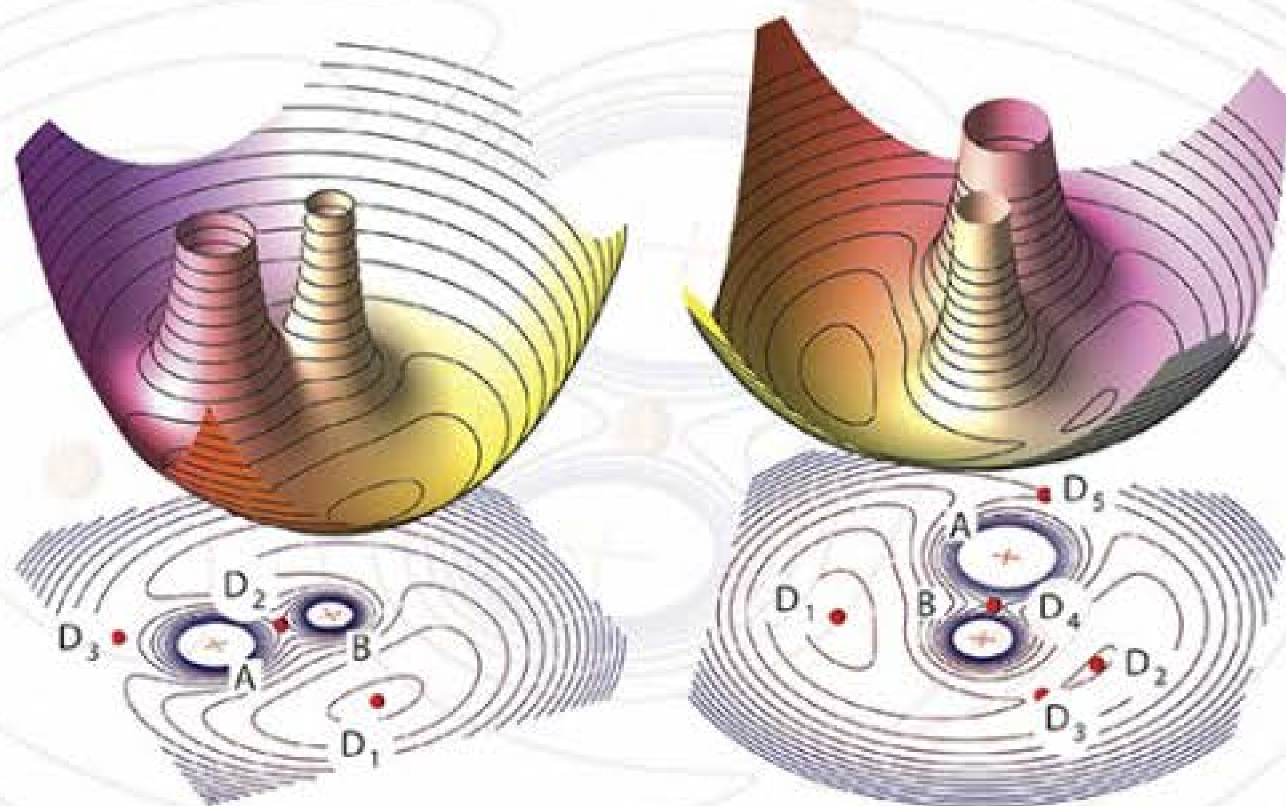
ROUBEN ROSTAMIAN

Programming Projects in C
for Students of Engineering, Science, and Mathematics



siam
Computational Science & Engineering

Programming Projects in C

for Students of Engineering, Science, and Mathematics

Computational Science & Engineering

The SIAM series on Computational Science and Engineering publishes research monographs, advanced undergraduate- or graduate-level textbooks, and other volumes of interest to an interdisciplinary CS&E community of computational mathematicians, computer scientists, scientists, and engineers. The series includes both introductory volumes aimed at a broad audience of mathematically motivated readers interested in understanding methods and applications within computational science and engineering and monographs reporting on the most recent developments in the field. The series also includes volumes addressed to specific groups of professionals whose work relies extensively on computational science and engineering.

SIAM created the CS&E series to support access to the rapid and far-ranging advances in computer modeling and simulation of complex problems in science and engineering, to promote the interdisciplinary culture required to meet these large-scale challenges, and to provide the means to the next generation of computational scientists and engineers.

Editor-in-Chief

Donald Estep
Colorado State University

Editorial Board

Omar Ghattas
University of Texas at Austin

Max Gunzburger
Florida State University

Des Higham
University of Strathclyde

Michael Holst
University of California, San Diego

David Keyes
Columbia University and KAUST

Max D. Morris
Iowa State University

Alex Pothen
Purdue University

Padma Raghavan
Pennsylvania State University

Karen Willcox
Massachusetts Institute of Technology

Series Volumes

Rostamian, Rouben, *Programming Projects in C for Students of Engineering, Science, and Mathematics*

Smith, Ralph C., *Uncertainty Quantification: Theory, Implementation, and Applications*

Dankowicz, Harry and Schilder, Frank, *Recipes for Continuation*

Mueller, Jennifer L. and Siltanen, Samuli, *Linear and Nonlinear Inverse Problems with Practical Applications*

Shapira, Yair, *Solving PDEs in C++: Numerical Methods in a Unified Object-Oriented Approach, Second Edition*

Borzì, Alfio and Schulz, Volker, *Computational Optimization of Systems Governed by Partial Differential Equations*

Ascher, Uri M. and Greif, Chen, *A First Course in Numerical Methods*

Layton, William, *Introduction to the Numerical Analysis of Incompressible Viscous Flows*

Ascher, Uri M., *Numerical Methods for Evolutionary Differential Equations*

Zohdi, T. I., *An Introduction to Modeling and Simulation of Particulate Flows*

Biegler, Lorenz T., Ghattas, Omar, Heinkenschloss, Matthias, Keyes, David, and van Bloemen Waanders, Bart, Editors, *Real-Time PDE-Constrained Optimization*

Chen, Zhangxin, Huan, Guanren, and Ma, Yuanle, *Computational Methods for Multiphase Flows in Porous Media*

Shapira, Yair, *Solving PDEs in C++: Numerical Methods in a Unified Object-Oriented Approach*

ROUBEN ROSTAMIAN
University of Maryland, Baltimore County
Baltimore, Maryland

Programming Projects in C

for Students of Engineering, Science, and Mathematics

siam®
Society for Industrial and Applied Mathematics
Philadelphia

Copyright © 2014 by the Society for Industrial and Applied Mathematics

10 9 8 7 6 5 4 3 2 1

All rights reserved. Printed in the United States of America. No part of this book may be reproduced, stored, or transmitted in any manner without the written permission of the publisher. For information, write to the Society for Industrial and Applied Mathematics, 3600 Market Street, 6th Floor, Philadelphia, PA 19104-2688 USA.

Trademarked names may be used in this book without the inclusion of a trademark symbol. These names are used in an editorial context only; no infringement of trademark is intended.

Intel is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

Linux is a registered trademark of Linus Torvalds.

Mac is a trademark of Apple Computer, Inc., registered in the United States and other countries. *Programming Projects in C for Students of Engineering, Science, and Mathematics* is an independent publication and has not been authorized, sponsored, or otherwise approved by Apple Computer, Inc.

Maple is a trademark of Waterloo Maple, Inc.

MATLAB is a registered trademark of The MathWorks, Inc. For MATLAB product information, please contact The MathWorks, Inc., 3 Apple Hill Drive, Natick, MA 01760-2098 USA, 508-647-7000, Fax: 508-647-7001, *info@mathworks.com*, *www.mathworks.com*.

PostScript is a registered trademark of Adobe Systems Incorporated in the United States and/or other countries.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Windows is a registered trademark of Microsoft Corporation in the United States and/or other countries.

Figures 15.1 (right image) and 15.2 courtesy of Stockvault.

Figure 19.2 courtesy of the Library of Congress.

Library of Congress Cataloging-in-Publication Data

Rostamian, Rouben, 1949-
Programming projects in C for students of engineering, science, and mathematics / Rouben Rostamian.
pages cm. – (Computational science and engineering series ; 13)
Includes bibliographical references and index.
ISBN 978-1-611973-49-5
1. Science–Data processing. 2. Engineering–Data processing. 3. Mathematics–Data processing. 4. C (Computer program language) I. Title.
Q183.9R67 2014
502.85'5133–dc23
2014012614

siam is a registered trademark.

Contents

Chapter interdependencies xi

Preface xiii

I A common background 1

1 Introduction 3

1.1 An overview of the book . . . 3
1.2 Why C? . . . 3
1.3 Which version of C? . . . 4
1.4 Operating systems . . . 7
1.5 The compiler and other software . . . 7
1.6 Interfaces and implementations . . . 9
1.7 Advice on writing . . . 11
1.8 Special notations . . . 11

2 File organization 13

3 Streams and the Unix shell 15

4 Pointers and arrays 19

4.1 Pointers . . . 19
4.2 Pointer types . . . 20
4.3 The pointer to void . . . 20
4.4 Arrays . . . 21
4.5 Multidimensional arrays . . . 23
4.6 Strings . . . 24
4.7 The command-line arguments . . . 25

5 From strings to numbers 27

5.1 The function `strtod()` . . . 27
5.2 The function `strtol()` . . . 28
5.3 The functions `atof()`, `atol()`, and friends . . . 29

6 Make 31

6.1 Multifile programs . . . 31
6.2 Separate compilation and linking . . . 31
6.3 File dependencies . . . 32

6.4 *Makefile* version 1 . . . 34
6.5 How to run `make` . . . 35
6.6 *Makefile* version 2 . . . 35
6.7 *Makefile* version 3 . . . 36
6.8 *Makefile*: The final version . . . 38
6.9 Linking with external libraries . . . 40
6.10 Multiple executables in one *Makefile* . . . 40

II Projects **43**

7 Allocating memory: `xmalloc()` **45**
7.1 Introduction . . . 45
7.2 A review of `malloc()` . . . 45
7.3 The program . . . 48
7.4 The interface and the implementation . . . 49
7.5 *Project Xmalloc* . . . 52

8 Dynamic memory allocation for vectors and matrices: *array.h* **53**
8.1 Introduction . . . 53
8.2 Constructing vectors of arbitrary types . . . 54
8.3 A scheme for dynamically allocated matrices . . . 56
8.4 Constructing matrices of arbitrary types . . . 57
8.5 *Project array.h* . . . 59

9 Reading lines: `fetch_line()` **63**
9.1 Introduction . . . 63
9.2 Reading one line at a time with `fgets()` . . . 63
9.3 Trimming whitespace and comments . . . 65
9.4 The program . . . 66
9.5 The files *fetch-line.[ch]* . . . 67
9.6 *Project fetch_line* . . . 70

10 Generating random numbers **71**
10.1 The `rand()` and `srand()` functions . . . 71
10.2 Bitmap images . . . 73
10.3 The program . . . 74
10.4 The file *random-pbm.c* . . . 75
10.5 *Project Random Bitmaps* . . . 78

11 Storing sparse matrices **79**
11.1 Introduction . . . 79
11.2 The CCS format . . . 80
11.3 The program . . . 81
11.4 The files *sparse.[ch]* . . . 81
11.5 *Project Sparse Matrix* . . . 82

12 Sparse systems: The UMFPACK library **85**
12.1 Introduction . . . 85
12.2 The basics . . . 85
12.3 The program . . . 86

12.4 *umfpack-demo1.c* . . . 87
12.5 *umfpack-demo2.c* . . . 89
12.6 *umfpack-demo3.c* and the triplet form . . . 90
12.7 *Project* UMFPACK . . . 92

13 Haar wavelets 93
13.1 A brief background . . . 93
13.2 The space $L^2(0,1)$ . . . 93
13.3 Haar's construction . . . 94
13.4 The decomposition $V_j = V_{j-1} \oplus W_{j-1}$ . . . 97
13.5 From functions to vectors . . . 98
13.6 The Haar wavelet transform . . . 100
13.7 Functions of two variables . . . 102
13.8 An overview of the *wavelet* module . . . 104
13.9 The file *wavelet.h* . . . 104
13.10 The file *wavelet.c* . . . 105
13.11 *Project Wavelets* . . . 109

14 Image I/O 113
14.1 Digital images and image file formats . . . 113
14.2 Bitmaps and the PBM image format . . . 114
14.3 Grayscale images and the PGM image format . . . 116
14.4 Color images and the PPM image format . . . 117
14.5 The *libnetpbm* library . . . 118
14.6 A no-frills demo of *libnetpbm* . . . 121
14.7 The interface of the *image-io* module . . . 121
14.8 The implementation of the *image-io* module . . . 124
14.9 The file *image-io-test-0.c* . . . 129
14.10 *Project Image I/O* . . . 131

15 Image analysis 135
15.1 Introduction . . . 135
15.2 The truncation error in a grayscale image . . . 137
15.3 The truncation error in a color image . . . 138
15.4 Image reconstruction . . . 138
15.5 The program . . . 139
15.6 The implementation of *image-analysis.c* . . . 139
15.7 *Project Image Analysis* . . . 144

16 Linked lists 147
16.1 Linked lists . . . 147
16.2 The program . . . 148
16.3 The function `ll_push()` . . . 148
16.4 The function `ll_pop()` . . . 150
16.5 The function `ll_free()` . . . 151
16.6 The function `ll_reverse()` . . . 152
16.7 The function `ll_sort()` . . . 153
16.8 The function `ll_filter()` . . . 157
16.9 The function `ll_length()` . . . 159
16.10 *Project Linked Lists* . . . 159

17 The evolution of species 161

17.1 Introduction 161
17.2 A more detailed description 162
17.3 The *World Definition File* 165
17.4 The program's user interface 166
17.5 The program's components 167
17.6 The file *evolution.h* 168
17.7 The files *read.[ch]* 169
17.8 The files *write.[ch]* 174
17.9 The files *world-to-eps.[ch]* 175
17.10 Interlude (and a mini-project) 176
17.11 The file *evolution.c* 177
17.12 Experiments 188
17.13 Animation 190
17.14 *Project Evolution* 191

18 The Nelder–Mead downhill simplex 193

18.1 Introduction 193
18.2 The algorithm 193
18.3 Problems with the Nelder–Mead algorithm 197
18.4 An overview of the program 198
18.5 The interface 198
18.6 The implementation 200
18.7 *Project Nelder–Mead*: Unconstrained optimization 207
18.8 Constrained optimization 211
18.9 *Project Nelder–Mead*: Constrained optimization 212
18.10 Appendix: Orthogonal projection onto $A\boldsymbol{x} = \boldsymbol{b}$ 213

19 Trusses 215

19.1 Introduction 215
19.2 One-dimensional elasticity 217
19.3 From energy to force 221
19.4 The energy of a truss 222
19.5 From energy to equilibrium 223
19.6 The *Truss Description File* (*TDF*) 224
19.7 An overview of the program 226
19.8 The interface 227
19.9 Reading and writing: *truss-io.[ch]* 230
19.10 The files *truss-to-eps.[ch]* 240
19.11 Interlude (and a mini-project) 241
19.12 The file *truss.c* 242
19.13 The file *truss-demo.c* 247
19.14 *Project Truss* 249

20 Finite difference schemes for the heat equation in one dimension 251

20.1 The basic idea of finite differences 251
20.2 An explicit scheme for the heat equation 253
20.3 An implicit scheme for the heat equation 256
20.4 The Crank–Nicolson scheme for the heat equation 258
20.5 The Seidman sweep scheme for the heat equation 260

20.6 Test problems 263
20.7 The program 266
20.8 The files *problem-spec.[ch]* 267
20.9 The file *heat-implicit.c* 272
20.10 *Project Finite Differences in One Dimension* 280

21 The porous medium equation 283
21.1 Introduction 283
21.2 Barenblatt's solution 283
21.3 Generalizations 284
21.4 The finite difference scheme 285
21.5 The program 286
21.6 The files *problem-spec.[ch]* 288
21.7 The file *pme-seidman-sweep.c* 288
21.8 *Project Porous Medium* 289
21.9 Appendix: The porous medium equation as a population dynamics model 289

22 Gaussian quadrature 291
22.1 Introduction 291
22.2 Lagrange interpolation 292
22.3 Legendre polynomials 294
22.4 The Gaussian quadrature formula 295
22.5 The program 296
22.6 The files *gauss-quad.[ch]* 297
22.7 *Project Gaussian Quadrature* 299

23 Triangulation with the *Triangle* library 301
23.1 Introduction 301
23.2 The program 302
23.3 The file *problem-spec.h* 303
23.4 The file *problem-spec.c* 305
23.5 The files *mesh.h* and *mesh.c* 311
23.6 The file *mesh-demo.c* 313
23.7 Installing *Triangle* 315
23.8 *Project Triangulate* 316

24 Integration on triangles 319
24.1 Introduction 319
24.2 The Taylor, Wingate, and Bos (TWB) quadrature 321
24.3 The files *twb-quad.[ch]* 322
24.4 The program 326
24.5 The files *plot-with-geomview.[ch]* 328
24.6 Modifying the file *problem-spec.c* 329
24.7 The file *twb-quad-demo.c* 330
24.8 *Project TWB Quadrature* 334

25 Finite elements 337
25.1 The Poisson equation 337
25.2 The weak formulation 338
25.3 The Galerkin approximation 340

25.4 An overview of the FEM 341
25.5 Error analysis 345
25.6 The program 347
25.7 Changes in *problem-spec.c* 349
25.8 The file *poisson.h* 350
25.9 The file *poisson.c* 351
25.10 The file *fem-demo.c* 358
25.11 Further reading 359
25.12 *Project FEM 1* 360

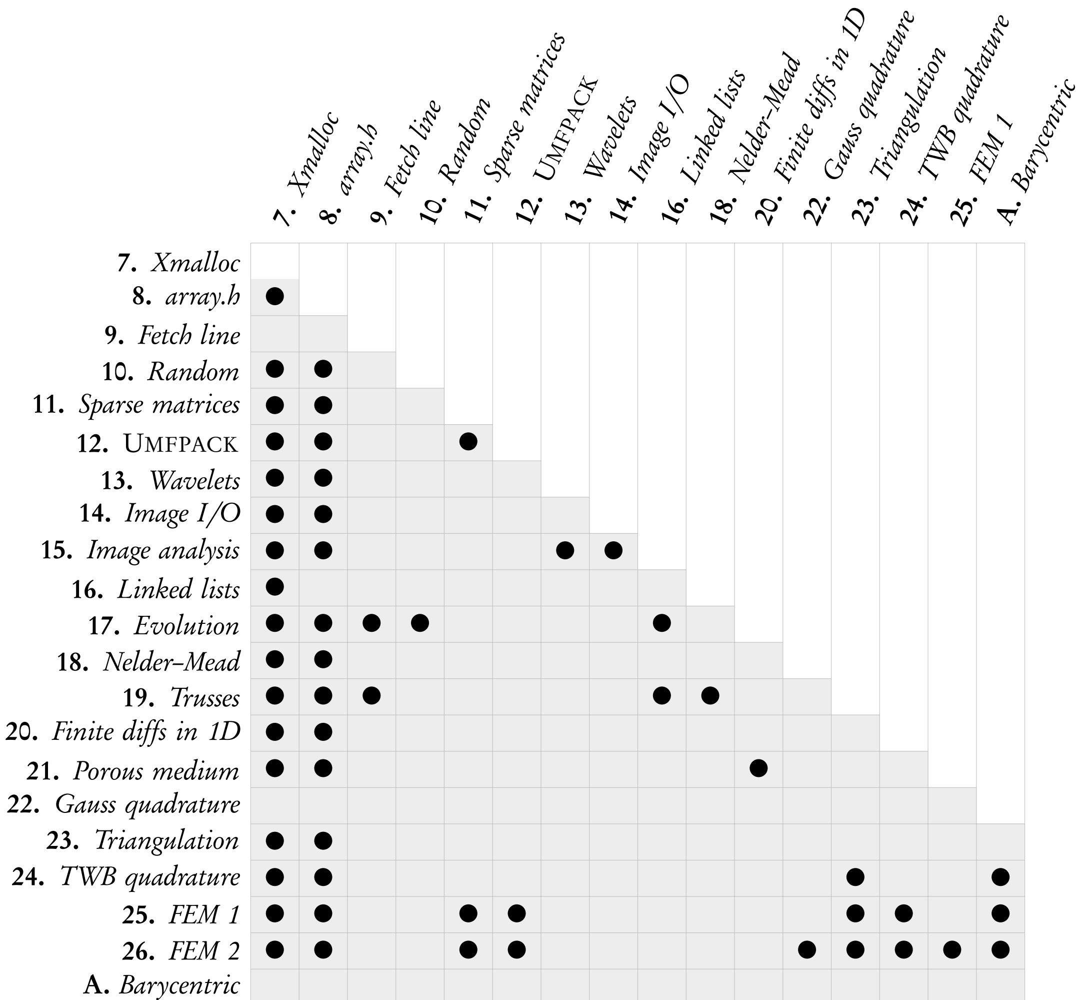
26 Finite elements: Nonzero boundary data 361
26.1 The problem 361
26.2 The weak formulation 362
26.3 The Galerkin approximation 364
26.4 The program 366
26.5 The file *problem-spec.[ch]* 366
26.6 The file *poisson.c* 369
26.7 *Project FEM 2* 373

A Barycentric coordinates 375
A.1 Barycentric coordinates 375
A.2 Calculus on a triangle 377

Bibliography 381

Index 387

Chapter interdependencies



	7. *Xmalloc*	8. *array.h*	9. *Fetch line*	10. *Random*	11. *Sparse matrices*	12. UMFPACK	13. *Wavelets*	14. *Image I/O*	16. *Linked lists*	18. *Nelder–Mead*	20. *Finite diffs in 1D*	22. *Gauss quadrature*	23. *Triangulation*	24. *TWB quadrature*	25. *FEM 1*	A. *Barycentric*
7. *Xmalloc*																
8. *array.h*	●															
9. *Fetch line*																
10. *Random*	●	●														
11. *Sparse matrices*	●	●														
12. UMFPACK	●	●			●											
13. *Wavelets*	●	●														
14. *Image I/O*	●	●														
15. *Image analysis*	●	●					●	●								
16. *Linked lists*	●															
17. *Evolution*	●	●	●	●					●							
18. *Nelder–Mead*	●	●														
19. *Trusses*	●	●	●						●	●						
20. *Finite diffs in 1D*	●	●														
21. *Porous medium*	●	●									●					
22. *Gauss quadrature*																
23. *Triangulation*	●	●														
24. *TWB quadrature*	●	●											●			●
25. *FEM 1*	●	●			●	●							●	●		●
26. *FEM 2*	●	●			●	●						●	●	●	●	●
A. *Barycentric*																

To find the prerequisites of a chapter, find the chapter title along the left edge, go across horizontally to the bullet marks, and then go vertically to the prerequisite chapters. For instance, **Chapter 23:** *Triangulation* depends on **Chapter 7:** *Xmalloc*, and **Chapter 8:** *array.h*. Chapters prior to Chapter 7 are not listed; these provide a general background for the entire book and should be considered prerequisites for everything.

Preface

This book is written for graduate and advanced undergraduate students of sciences, engineering, and mathematics as a tutorial on how to think about, organize, and implement programs in scientific computing. It may be used as a textbook for classroom instruction, or by individuals for self-directed learning. It is the outgrowth of a course that I have taught periodically over nearly 20 years at UMBC. In the beginning it was targeted to graduate students in Applied Mathematics to help them quickly acquire programming skills to implement and experiment with the ideas and algorithms mostly related to their doctoral researches. Over the years it has gained popularity among the Mechanical Engineering students. In recent years, about a quarter of the enrollment has come from the College of Engineering. Additionally, I have had the pleasure of having a number of advanced undergraduate student in the course; they have done quite well.

The course's, and by extension the book's, immediate goal is to provide an interesting and instructive set of problems—I call them *Projects*—each of which begins with the presentation of a problem and an algorithm for solving it and then leads the reader through implementing the algorithm in C and compiling and testing the results. The ultimate goal in my mind, however, is pedagogy, not programming per se. Most students can attest that there is a substantial gap between what one learns in an undergraduate course dedicated to programming and what is required to implement ideas and algorithms of scientific computing in a coherent fashion. This book aims to bridge that gap through a set of carefully thought-out and well-developed programming projects. The book does not "lecture" the reader; rather, it shows the way—at times by doing, and at times by prompting what to do—to lead him/her toward a goal. Paramount in my objectives is to instill a habit of, and an appreciation for, *modular program organization*. Breaking a large program into small and logically independent units makes it easier to understand, test/debug, and alter/expand, and—as demonstrated abundantly throughout—it makes the parts available for reuse elsewhere.

I hope that the reader will take away more than just programming techniques from this book. I have strived to make the projects interesting, intriguing, inviting, challenging, and illuminating on their own, apart from their programming aspects. The range of the topics inevitably reflects my tastes, but I hope that there is enough variety here to enable any reader to find several rewarding projects to work on. Some of my favorite projects are

- the *Nelder–Mead simplex algorithm* for minimizing functions in $\boldsymbol{R}^n$ (with or without constraints) with applications to computing finite deformations of trusses under large loads via minimizing the energy;
- the *Haar wavelet transform* in one and two dimensions, with applications to image analysis and image compression;

- a very simple yet intriguing model of *evolution through natural selection* and the effect of the environment on the emergence of genetically distinct species (speciation);
- the comparison/contrast of several *finite difference algorithms* for solving the time-dependent linear heat equation and extending one of the algorithms to solving the (nonlinear and degenerate) porous medium equation; and
- a minimal implementation of the *finite element method* for solving second order elliptic partial differential equations on arbitrary two-dimensional domains through unstructured triangular meshes and linear elements.

Additionally, I am particularly pleased with the *array.h* header file of Chapter 8 which provides a set of preprocessor macros for allocating and freeing memory for vectors and matrices of *arbitrary types* entirely within the bounds of standard C. That header file is used throughout the rest of the book.

To reach the book's intended readership, that is, the advanced undergraduate through beginning graduate students, I have made a consistent effort throughout to keep the mathematical prerequisites and jargon to a minimum and have not shunned from skirting technical issues to the extent that I could. For instance, the Galerkin approximation and the finite element method are introduced in Chapter 25 without explicit references to Hilbert or Sobolev spaces, although these concepts are brought up in the subsequent chapter. I have included plenty of references to the literature to help the curious reader to learn more. I believe that a good knowledge of undergraduate multivariable calculus and linear algebra is all that is needed to follow the topics in this book, but a graduate student's technical maturity certainly will help.

Part I of the book, consisting of Chapters 1 through 6, is a prerequisite for Part II, which makes up the rest of the book. A working familiarity with the concepts introduced in Part I is tacitly assumed throughout. The chapters of Part II consist of individual projects. Part II is *definitely not* intended for linear/sequential reading. A chart on page xi shows the chapter interdependencies. Chapter 18 on the Nelder–Mead simplex method, for instance, depends on Chapters 7 (memory allocation) and Chapter 8 (vectors and matrices). The way to read this book, therefore, is to pick a topic, look up its prerequisite in the chart, and then sequence your reading accordingly.

For classroom teaching, I select topics that reflect the interests of the majority of the class, which vary from semester to semester. The most recent semester's syllabus consisted of, in the order of coverage, the following:

- Chapter 7: allocating memory
- Chapter 8: constructing vectors and matrices
- Chapter 18: minimization through the Nelder–Mead simplex method
- Chapter 14: reading and writing digital images
- Chapter 23: using the *Triangle* library to triangulate two-dimensional polygonal domains
- Appendix A: an introduction to barycentric coordinates
- Chapter 24: integration over a triangulated domain
- Chapter 11: storage methods for sparse matrices
- Chapter 12: solving sparse linear systems using the UMFPACK library
- Chapter 25: a finite element method for solving the Poisson equation with zero Dirichlet boundary conditions
- Chapter 22: Gaussian quadrature

- Chapter 26: second order elliptic partial differential equation with arbitrary Dirichlet and Neumann boundary conditions

Naturally the syllabus and its pace should be adjusted to what the students can handle. Parts marked [optional] in a chapter's *Projects* section provide exercises that go beyond minimal learning objectives. Most students voluntarily carry out all the parts, regardless of the [optional] tags.

I should emphasize that neither the course nor this book is a primer on C. Most of my students have had at least one semester of an undergraduate course in C or a C-like low-level procedural programming language, although there have been a few whose prior programming experience has been nothing but MATLAB®, and they have succeeded through hard work, self-study, and some help from me and their classmates. In class I don't dwell on the basics of C programming. I do, however, devote time to pointing out the more subtle programming issues in anticipation of questions that may arise in particular projects. The topics in Part I reflect some of those class presentations. For a self-study, refresher, and reference on the C programming language I recommend Kochan's book [35].

Every program in this book is in full conformance with the 1999 ISO standard C, also known as C99. With minor changes, pointed out in Section 1.3, you may revert them to the 1989 standard, C89, if you so wish. The latest C standard, C11, was announced in 2011, but as of this writing there are no C11 compilers that fully support it; therefore I have avoided special features that were introduced in C11. See Section 1.3 for more on this.

Some of the projects call for supplementary files, mostly consisting of programs or program fragments, which may be obtained from the book's website at

`<www.siam.org/books/cs13/>`.

These supplemental programs are not difficult per se but may require specialized knowledge, such as the detailed syntax of the *PostScript* language or the interface to the *Triangle* library, which I do not wish to make prerequisites for completing the projects. The website also includes additional information, animations/demos, and other miscellany which may be of help with completing the projects.

It is customary in a book's preface to thank those who have been instrumental in bringing the book about. For the present book my thanks go first and foremost to the scores of students who have, over the years, put up with the loose sheets of printed paper which I have distributed weekly in class in lieu of a conventional textbook. I trust that having this book in hand will make for a less stressful—even pleasurable, I hope—learning experience. I am also indebted to the anonymous reviewers whose many constructive ideas and suggestions have been incorporated into the current presentation. Finally, my heartfelt thanks go to SIAM's amazing staff whose enthusiastic support and expert advice in all phases of this book's production have improved the original manuscript by an order of magnitude.

Rouben Rostamian
UMBC, March 2014

Chapter 1

Introduction

1.1 ▪ An overview of the book

This is a somewhat unusual computing/programming book in that essentially all of its programs are presented in incomplete fragments. That is by design. You, the reader, are charged with completing the programs by following the instructions and outlines in each case, and thus developing your skills in programming scientific computing algorithms. In that sense, this book serves a purpose similar to that of a book of *études* for a pianist. You learn by doing.

Part I, consisting of Chapters 1 through 6, brings together a set of diverse topics that form a part of the minimal working background for the rest of the book. If you feel familiar with that material, skim through it just to be certain. If the material is new to you, then read through those chapters patiently and internalize the concepts as much as possible. As you work on the projects in Part II, you may want to revisit Part I from time to time to reinforce your understanding of those topics.

Part II, which makes up the rest of the book, consists of *Projects*, one per chapter, of a diverse collection of topics. Each project begins with the presentation of a problem and an algorithm for solving it and then leads the reader through implementing the algorithm in C. Every project comes with an outline that shows how the program is broken into multiple files, and how each file is broken into individual functions, mostly by giving the functions' prototypes. The C code for the more difficult/tricky parts of the project is often given in full. You are asked to supply the rest.

Part II is *definitely not* intended for linear/sequential reading. A chart on page xi shows the chapter interdependencies. Pick a project that interests you, consult that chart to determine its prerequisites, and then start with the very first prerequisite and make your way forward.

There is a delicate balance between providing too much versus too little information if a project is to be a worthwhile learning exercise without causing undue frustration. I have strived to achieve that balance based on my experiences in the classroom.

1.2 ▪ Why C?

I have to admit that I have no good answer to the question "Why C?" Come to think of it, I have no good answer to the question "Why English?" either. I have written this book in English because that's the language in which I feel most comfortable expressing my

ideas. I can say the same thing about C. But to be less flip about it, C has many things going for it. Let me itemize:

- C is a relatively old and well-established programming language. It dates back to the early 1970s. C compilers have been available essentially for free on every platform since its inception. That and Kernighan and Ritchie's superbly lucid exposition [31] were responsible for propelling the language to widespread popularity and permanence. With C, you are not investing your time and effort in the faddish language of the day.

- After evolving for several years, C was eventually standardized as ANSI C in 1989 and later adopted as a worldwide ISO standard. That ANSI/ISO version of C, known as C89, is widely supported on all computing platforms. Programs written according to the standard are assured to be portable across all platforms. Kernighan and Ritchie's second edition [32] reflected the C89 standard and helped continue the spread of the language and its popularity. See Section 1.3 regarding C's more recent developments.

- The use of C in scientific computing is of a rather recent origin. C was conceived as a low-level programming language—just a small step up from an assembly language—whose initial applications were in writing operating systems and basic command-line interface tools. An abstraction layer isolates C from the vicissitudes of the underlying hardware and allows one to write portable code. Nevertheless, C remains "close to the metal", especially with respect to its memory management. The language's low-level nature imposes a somewhat steep learning curve—that's one of its drawbacks—but it rewards the patient learner with a sense of absolute control and a total power over the hardware. In many cases one can almost see how the algorithm is reduced to shuffling data in the computer's memory. With some extra care, one can detect and eliminate wasteful operations. Generally it is easier to write efficient code in C than it is in higher-level languages where the connection with the hardware tends to be obscured by multiple layers of abstractions.

  And ultimately, the joy of writing a program where you retain total control and can see the minutest details from the ground up cannot be overestimated.

- C has the advantage of being a small language. It is not an exaggeration to claim that one with some prior experience with programming languages can comfortably learn most of the C in less than one week. If you go through about half of this book's projects, it is likely that you will have used nearly 90% of the C language and a good part of the associated *standard library*. At the same time, one has to acknowledge that a small language has its drawbacks: sometimes it takes quite a bit of work to accomplish seemingly trivial tasks since the language lacks built-in "big tools". You, the user, are expected to build your own "big tools" from the miniature components that the language provides. That may sound disheartening at first, but it is not as bad as it may appear. True, to build a general utility for allocating and freeing memory for vectors and matrices takes up two chapters in this book, but you do that only once, as I did 30 years ago; then you use it an innumerable number of times afterward.

1.3 ▪ Which version of C?

I sketched the early history of the development of C up to C89 in the previous section. A new ISO standard C, known as C99, was announced in 1999. It introduced major

extensions and enhancements relative to C89, some of which, such as the complex data type (as in $z = x + iy$), are vital to scientific computing, although we have no use for them in this book. (The old C89 lacks a complex data type.) Despite that, the reaction of the C programmer community to C99 was mostly cold, and not infrequently hostile. C99 was, and still is, perceived as deviating from C's original minimalist philosophy. The latest C standard, announced in 2011 and known as C11, addresses the objections by relegating some of the most disputed C99 extensions to optional (i.e., not required) status. There has been no great rush to embrace C11; the C programming world tends to be quite conservative in this regard. C89 has such a strong foothold within the C programming community that, after almost 25 years (as of this writing) and the emergence of two newer standards, C89 is still viewed by many as "the one true C".

The programs in this book take advantage of some of the more useful (and noncontroversial) features introduced in C99, such as the //-style comments and structure initializers with named members. *All of the book's programs are designed to conform fully to the C99 standard.* Failures in conformance, if any, are bugs, and I should take the blame for them.

To those readers who wish to stay with the classic C89 version of C, I must give the reassuring words that the infringements into the C99 territory, relative to C89, are by no means essential. It is quite trivial to revert this book's code to strict C89. To help you with that, should you have a desire to do so, here is the *complete list* of the C99-specific constructs that I have used in the book:

Comments: In C89, everything between a /* and a */ is taken as a comment and therefore skipped over. Such comments may span more than one line. C99 adds an alternative method of commenting: Everything from a // to the end-of-line is taken as a comment and is skipped over. Naturally such comments cannot span more than one line.[1] The two types of comments may coexist in a C99 program. In this book I use the //-style comments frequently since they take up less room on a printed page.

For-loops: In C89, a **for**-loop's index must be declared outside of the loop, as in

```
int i;
...
for (i = 0; i < 6; i++)
    whatever;
```

Upon a normal exit from that loop, the value of `i` will be 6. In C99 we have the option of declaring the index *within* the **for**-loop itself, as in

```
for (int i = 0; i < 6; i++)
    whatever;
```

Here the index `i` is local; it's not accessible outside the **for**-loop. I quite like this C99 innovation; there is no sense in polluting the rest of the code with `i` if there is no need for it. I have used the C99-style **for**-loops in quite a few places, but some C89-style **for**-loops remain. Old habits die hard.

[1] Actually this is not strictly correct; a //-style comment *may* be continued into the next line if the line's last character is \, but I have never felt the urge to use that feature.

Initializing structures: Consider the following structure:

```
struct mystruct {
    int n;
    double x;
    double y;
};
```

In C89 we may define and initialize an instance of this structure through:

```
struct mystruct S = { 12, 3.14, 2.78 };
```

The values `12`, `3.14`, and `2.78` are assigned to the members `n`, `x`, and `y` of `S`, respectively. Naturally, the numbers should be listed precisely in the order in which the target symbols `n`, `x`, and `y` appear in the structure's declaration.

In C99 we may define and initialize an instance of that structure in an alternative fashion:

```
struct mystruct S = { .n = 12, .x = 3.14, .y = 2.78 };
```

The order of the entries here is immaterial since each number is identified explicitly with the name of its target.

There hardly seems an advantage of one method over the other in such a simple case, but you may agree that in a structure with tens of members, C99's explicit version would be less confusing.

Even in the simple case above there is an advantage to the C99 version. Suppose the values of `n` and `y` are available at the initialization time but the value of `x` isn't. In the C99 syntax we do

```
struct mystruct S = { .n = 12, .y = 2.78 };        // x is not assigned
```

There is no way to do that in C89 other than by assigning a dummy value to `x`.

The `z` modifier in formatted printing: The proper way of printing the value of a `size_t` variable is through `printf()`'s `%zu` conversion flag, as in

```
size_t n = ...;
printf("n = %zu\n", n);
```

The `z` modifier was introduced in C99. The (almost) equivalent code in C89 requires a cast, as in

```
size_t n = ...;
printf("n = %lu\n", (unsigned long)n);
```

This assumes that `size_t` is equivalent to "**`unsigned long`**", which it likely is, but the C standard makes no such guarantee. Clearly the C99 way is the clean way of doing this.

Mixing declarations and code: C89 requires that all identifier declarations come before other code within a code block. (A *code block* is what is enclosed between *curly braces* `{` and `}`.) Thus, for instance, one declares all identifiers as the first thing in a function definition. C99 permits the mixing of declarations and code. I have adhered to the C89 requirement throughout most of the book, with only a few exceptions where I felt that a midblock declaration leads to a more expressive code. I have explicitly noted such exceptions where they occur. To revert to C89, just move those midblock declarations to the top of the block.

Inline functions: The purpose of the `inline` function specifier, introduced in C99, is to suggest to the compiler that the calls to a given function be as fast as possible. The compiler may honor or ignore the request. Generally the `inline` specifier makes sense in the context of small and intensely used functions. The one-liner function `random()` in Chapter 10, for instance, is declared `inline`. You may safely remove the `inline` specifier if you want a strict C89 code.

If you are not an expert C programmer, you should have a good C reference book at hand when reading the present book. I wish I could tell you to go get Kernighan and Ritchie's third edition, but unfortunately there is no such a thing. The second edition [32] dates back to 1989 and is showing its age, and of course it has no knowledge of C99. I am sure there are some excellent C programming textbooks on the market. Use your favorite if you have one. Otherwise, consider Kochan's book [35], which is reasonably good. Whatever you do, I suggest that you stay away from web tutorials; they tend to be error-ridden and can lead you to form misunderstandings and develop bad habits which may be difficult to unlearn afterward.

1.4 ▪ Operating systems

A C program does not live in the abstract; eventually you will compile and execute it on some computer. Although a standard-conforming program is platform independent, the way the program is compiled and executed depends very much on the platform, by which I mean the computer's operating system and its user interface. The most popular computer platforms nowadays are *Windows* produced by *Microsoft*, OS X for Mac computers produced by *Apple*, and the Unix operating system or its look-alikes, such as Linux, which can run on just about any computer hardware.

Since the thrust of this book is writing standard-conforming programs, the target platform is immaterial as far as the programs' contents go. However, I am forced to resort to a specific platform in order to demonstrate how the programs are *used*. I have chosen to do all such demonstrations in the context of a command-line based Unix terminal mainly because that is what I myself use, and also because the command-line exposes everything there is to know about running a program; there are no menus or icons that potentially can hide information.

If you are a Mac user, you may already know that the OS X operating system is a variant of Unix; therefore you gain full access to the Unix command-line utilities by bringing up its *Terminal* application. This book's Unix command-line examples should work on a Mac without change.

If you are a Windows user, you may install *Cygwin*, which provides a Unix-like environment within Windows. Then you should be able to compile, execute, and experiment with most of this book's programs. Some obstacles will remain, such as coaxing a large utility such as *Geomview* to work under Cygwin. My students find it simpler to install Linux alongside Windows on a separate partition and use that to gain unfettered access to everything they need to carry out their projects.

1.5 ▪ The compiler and other software

It goes without saying that you need a C compiler to compile your programs. If your platform comes with a native C compiler, then use it. Otherwise you will need to download and install a C compiler. I use the GNU C compiler (almost always) and Intel's C compiler (just for testing) on Linux, and Sun's C compiler on Solaris. See what is available for yours. It is difficult to be more specific here.

Read your compiler's documentation. Generally a compiler is invoked on the command line with a set of flags that determine the details of its behavior. I invoke the GNU C compiler as

```
gcc -Wall -pedantic -std=c99 -O2 files-to-compile
```

which compiles according to the C99 language specification and asks it to issue warnings if the code deviates from it. Change `-std=c99` to `-std=c89` to compile according to C89. The `-O2` flag (that's an *uppercase letter* "O") asks it to produce optimized machine code. See Chapter 6 on how to automate the compilation process.

In addition to a compiler, many projects call for specialized third-party software which you will need to download and install as needed. These are the following:

***Geomview*:** This is a utility for displaying and examining images of surfaces in three dimensions on your computer screen. We use *Geomview* to display the solutions of partial differential equations in chapters dealing with finite differences and finite elements. *Geomview* is a free and open software. On most Linux distributions you may install a precompiled *Geomview* package through a few mouse clicks. On a Mac, you may get *Geomview* from `<http://www.macports.org/>`. There is no native implementation for Windows, but you may run *Geomview* in Windows under *Cygwin* as noted earlier. Go to `<http://www.geomview.org/>` for instructions. You may even get *Geomview*'s source from there and compile it on your computer yourself, although that is not quite a trivial task due to dependencies on a slew of other program libraries.

Viewing an image in *Geomview* is as simple as typing "`geomview file.gv`" on the command-line.

Remark 1.1. ***Paraview*** from `<http://www.paraview.org/>` provides a functionality similar to *Geomview*, and like *Geomview*, it is a free and open software. An advantage over *Geomview* is its availability on all common computing platforms, i.e., Mac, Unix, and Windows. I have experimented with *Paraview* a bit but prefer *Geomview*, perhaps because of my familiarity with it.

Triangle: This is a state-of-the-art utility for triangulating two-dimensional polygonal domains. You may download the software from its source at

`<http://www.cs.cmu.edu/~quake/triangle.html>`,

but it won't work out of the box since you will need to read the instructions and set a few preprocessor options. I suggest that you get the slightly modified version from this book's website at `<www.siam.org/books/cs13>`. My modifications amount to setting the necessary preprocessor options to make it compile into an object file rather than a stand-alone program. The triangulation code itself is not touched.

UMFPACK**:** This is a state-of-the-art library for solving linear systems of equations, as in $Ax = b$, where A is an $n \times n$ sparse matrix. We use UMFPACK to solve the systems of equations that arise in our finite element implementations. On most Linux distributions you may install the UMFPACK library through a few mouse clicks. Look for a package named `libsuitesparse-dev` or `umfpack-dev`. On a Mac you may get UMFPACK from `<http://www.macports.org/>`. You may even get

UMFPACK's source from its author's website

<http://www.cise.ufl.edu/research/sparse/umfpack/>

and compile it yourself if you feel adventurous enough.

Netpbm: *Netpbm* is a suite of utilities and libraries for reading, writing, displaying, and manipulating images. Our image processing projects in Chapters 14 and 15 rely on the *Netpbm* library for image I/O. On Linux, installing the `netpbm` and `libnetpbm10-dev` packages will do. For a Mac, get *Netpbm* from

<http://www.macports.org/>.

An image viewer: The programs in Chapters 10, 14, and 15 produce images in the PGM and PPM formats. You will need a means of viewing such images on your computer screen. Just about any image viewing program should be able to recognize and handle these.

A PostScript viewer: The programs in Chapters 19 and 23 produce images in the *Encapsulated PostScript* (EPS) format. Viewing an EPS image requires specialized software. On Linux you may view an EPS image through the command-line as in "`evince file.eps`". It is likely that your Linux distribution installs `evince` automatically. If not, then get and install it, or use your own favorite EPS viewer, of which there are many. I have no specific recommendation for Mac. Search for an EPS viewer. You will find several.

1.6 ▪ Interfaces and implementations

It is a common practice to break up a large C program into multiple files, where the code in each file is responsible for performing a clearly defined and relatively simple task. Then a controlling unit, sometimes called the *driver*, ties the pieces together and makes a whole program. The benefit of splitting a program in this way is that smaller programming units are easier to understand, verify for correctness, alter, and debug.

In this book, a *module* refers to a set of program files, other than the driver, that as a whole serve to perform a well-defined task. For instance, the Nelder–Mead module of Chapter 18 finds the minima of a given function. The Nelder–Mead module is used in Chapter 19 to determine a truss's deformation by minimizing its energy. The word "module" is not an officially sanctioned term in C, although it is common in other programming languages. So I have taken the liberty of using it in the C context in this book.

As a program is broken into multiple files, the code within each file is broken into multiple functions, where each function performs a clearly defined and relatively simple task. Generally only a few—often just one or two—of the many functions within a file are meant to be visible to the outside world. The rest tend to be for internal use only within their own files, lending support to the other functions to accomplish the task which that particular file is designed to accomplish, but do not communicate with, and are not visible to, the outside. An experienced programmer marks such "internal use only" functions with a **`static`** declaration specifier. That tells the compiler and the linker that those functions have no visibility outside of their own files. This makes it possible for a program to have two totally different functions with *identical names* in two different files without risking conflict, as long as the functions are declared **`static`**. This is particularly significant in "real world" large projects where different parts of the program are written by different teams of programmers.

```c
#ifndef H_NELDER_MEAD_H
#define H_NELDER_MEAD_H

struct nelder_mead {
    double (*f)(...)
    int n
    double **s
    double *x
    double h
    double tol
    ...
};

int nelder_mead(
    struct nelder_mead *nm);

#endif /* H_NELDER_MEAD_H */
```

nelder-mead.h (the interface)

```c
#include "nelder-mead.h"

static void get_centroid(...)
{
    ...
}

static int done(...)
{
    ...
}

int nelder_mead(
    struct nelder_mead *nm)
{
    ...
}
```

nelder-mead.c (the implementation)

Figure 1.1: These fragments of the files *nelder-mead.h* and *nelder-mead.c*, extracted from Chapter 18's *Project Nelder–Mead*, illustrate the typical splitting of a program into an interface and an implementation.

The box on the right in Figure 1.1 shows fragments of the file *nelder-mead.c* from Chapter 18. Auxiliary functions `get_centroid()` and `done()` are declared **`static`** since they are meant for internal consumption within *nelder-mead.c*, while the function `nelder_mead()` is not declared **`static`** because it is the ultimate purpose of the file to make that function available to the outside world.[2]

The availability of the function `nelder_mead()` is announced through the *header file nelder-mead.h* shown in the left box in Figure 1.1. It provides a declaration of a "**`struct`** `nelder_mead`" structure, asserts that the function `nelder_mead()` takes a pointer to that structure as an argument, and returns an **`int`**.

The file *nelder-mead.h* is called *the interface* of the Nelder–Mead utility. The file *nelder-mead.c* is called *the implementation*. To connect with the Nelder–Mead utility, what a user[3] needs is the interface. The details of the operation that are hidden in the implementation are not the user's concern. (Think of "*No User Serviceable Parts Inside*" stamped on the implementation.) An upgrade from version 1 to version 2 of the Nelder–Mead utility may change the internals of the implementation, perhaps making it run more efficiently, but if the interface does not change, then the users won't have to change anything on their sides to reap the benefits.

A program's interface is analogous to a restaurant's menu, while the implementation is analogous to the restaurant's kitchen. The restaurant's patron interfaces with the menu and partakes of the food but has no business entering the kitchen. The kitchen staff has no "use" for the menu, but having a copy available in the kitchen can help coordinate the staff's work with what is advertised at the tables. For that same reason, it is an excellent

[2]If I were granted one wish to change C, I would make all functions **`static`** by default. A function with outside visibility—the technical term is *external linkage*—would be required to be declared so explicitly. But alas, we have to live with what we have.

[3]More often than not, that "user" is you. Once you are finished with implementing *nelder-mead.c*, you treat it at a "black box" that takes data and spits out results. You have become a "user" of *nelder-mead.c*.

idea to make a program's interface available to the implementation. That's one reason for having **#include**"nelder-mead.h" at the top of the file *nelder-mead.c* in Figure 1.1. Every properly written implementation should **#include** its own header file, if it has one. As to the **#ifndef** H_NELDER_MEAD_H, etc., appearing in *nelder-mead.h*, see the discussion of *#include guards* in Chapter 7.

Most programs you will encounter in this book come in **.h* and **.c* pairs, reflecting the interface and implementation paradigm. Hanson's book [26] is an excellent resource for learning about program organization in general, and interfaces and implementations in particular.

1.7 ▪ Advice on writing

In the interest of eliminating excessive whitespace within printed pages, I have removed almost all empty lines, and almost all comments, in the code samples shown. This is *definitely not* the way I write my own code in general. I use empty lines liberally to make the code as readable and appealing as possible. I preface *every function definition* with a good deal of comments explaining its purpose, the nature of its arguments, and the value that it returns. The comments may be obvious to me at the time of writing the function, but experience shows that the human memory is fallible and one needs all the help one can get from such comments when revisiting the code a few years hence.

Additionally, at the top of every file I identify its purpose and why it was written, and the date, and by whom. Identifying yourself as the author is particularly important; otherwise, 10 years down the road you will read the file again and will have no idea whether it is something that you wrote or someone else sent it to you. Documenting the authorship will save you the embarrassment (or worse) of claiming a code of being yours when it is not.

If I revise a file later on, I extend the comments by explaining when, how, and why it was revised. Such considerations may seem to be of secondary importance in the heat of the moment, but you will be thankful for having provided them years later when you refer to the code again.

1.8 ▪ Special notations

Throughout this book I have taken the liberty of typesetting certain C operators in special glyphs, merely for cosmetic appeal, as shown in the table below:

typewriter notation	<=	>=	!=	->
book's glyph	$\leq$	$\geq$	$\neq$	$\rightarrow$

Naturally, you will translate the book's glyphs into their conventional typewriter notations in your code.

Within the displayed code fragments, I have used the ▶ and ... markers to indicate a line of code that needs to be completed, e.g.,

```
▶ static void shrink(double **s, int n, int ia) ...
```

Mostly you will supply the elided code yourself, but I have provided the details in the more complex cases.

Chapter 2

File organization

The manner of organizing files and directories on one's computer is very much a matter of personal taste. I am not about to advise you to do what does not come naturally to you. Nevertheless, I am going to tell you how I organize *my* files—at least those that pertain to this book—in the hope that this may convey something worthwhile, especially in case you don't have strong preferences or prejudices in these matters.

My top-level directory is called *c-projects* since that's what this book is about. Feel free to call yours something else if you so wish. Under the *c-projects* directory I have individual subdirectories for each of the book's projects. Here is a partial listing:

```
$ cd c-projects
$ ls -F
nelder-mead/  vector-and-matrix/  xmalloc/
```

The "`$`" sign is the traditional symbol for the Unix shell prompt. You will see that throughout the book. The Unix command `ls` displays the list of files and subdirectories in the current directory. The `-F` flag tells `ls` to decorate directory names with trailing slashes. That helps to distinguish directories from files of other kind.

The Unix `mkdir` command makes a new directory:

```
$ mkdir gauss-quad
$ ls -F
nelder-mead/  gauss-quad/  vector-and-matrix/  xmalloc/
```

There is a considerable interdependence among this book's projects. For instance, the *xmalloc* module, whose code is developed under the *xmalloc* directory, is used in just about all other projects. So is the *array.h* header file, which is developed under the *vector-and-matrix* directory. We need to devise a strategy to make files developed under one directory available to a project in another directory. There are various ways of doing this, the absolute worst of which is to copy files from one directory to another. Keeping duplicate files on your computer *is evil*. You will edit one tomorrow and the other next week, and end up with two irreconcilable versions and much to be sorry about.

I do a *symbolic link* (called a *symlink* for short) instead of copying. For instance, the file *array.h* from the *vector-and-matrix* directory and the files *xmalloc.c* and *xmalloc.h* from the *xmalloc* directory are needed in the *nelder-mead* directory. So I do

```
1    $ cd nelder-mead
2    $ ln -s ../vector-and-matrix/array.h .
3    $ ln -s ../xmalloc/xmalloc.[ch] .
```

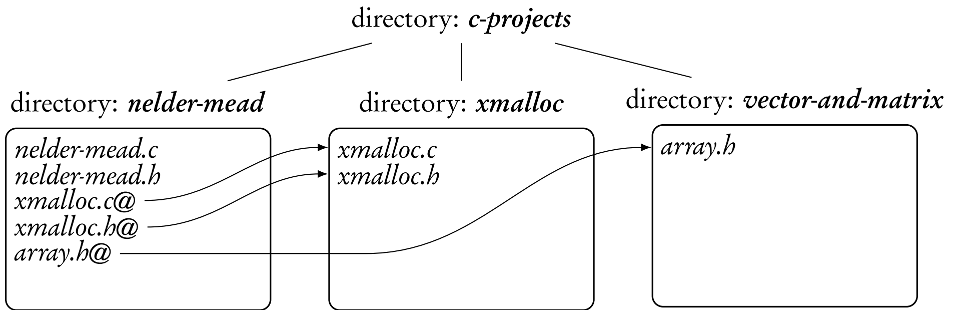


Figure 2.1: The files *xmalloc.c*, *xmalloc.h*, and *array.h* under the ***nelder-mead*** directory are symbolic links to files in the ***xmalloc*** and ***vector-and-matrix*** directories, as shown. The `-F` flag in `ls -F` decorates symbolic links with trailing "`@`" signs for displaying purposes. Beware that the "`@`" is *not* a part a file's name.

```
4   $ ls -F
5   array.h@  nelder-mead.c  nelder-mead.h xmalloc.c@ xmalloc.h@
```

The `-F` flag tells `ls` to decorate symbolic links with trailing "`@`" signs for displaying purposes; the "`@`" is *not* a part of the file's name. Don't overlook the dots at the ends of lines 2 and 3; they signify "the current directory" to the Unix shell. So those lines are really saying, "link files from such-and-such place into the current directory". Figure 2.1 shows the concept of symbolic links in a graphical way.

For most practical purposes, a symbolic link behaves like a real file, so having those symbolic links in the *nelder-mead* directory is as good as having the actual files there. Remember, however, that if you edit *xmalloc.c* that appears under *nelder-mead*, you are actually editing the file *../xmalloc/xmalloc.c*. There is only *one copy* of that file on the system, and that's a *good thing*; the one appearing under the *nelder-mead* directory is a mere "pointer" to the real one.

You will see symbolic links used extensively throughout this book for file organization purposes in the projects. Get into the habit of using them! They are available on Unix/Linux, Mac, and Windows.

Chapter 3

Streams and the Unix shell

A C program communicates with the outside world by reading from and writing to *streams*. A stream is an abstract concept; think of it as a pipe through which data flow. The program *opens* a stream for reading, or writing, and *closes* it when it's done. Three streams, called the `stdin`, `stdout`, and `stderr`, are opened automatically at the beginning of the execution of a C program—`stdin` for reading and the other two for writing. The program may open a number of additional streams.

There is a plethora of standard library functions for reading from streams. These include

`fgetc()` which reads one character at a time;
`fgets()` which reads one line at a time;
`fscanf()` which reads under the control of a format string; and
`ungetc()` which pushes a character back into a stream.

There are also `getchar()` and `scanf()` which are the specialized versions of `fgetc()` and `fscanf()` for reading from the `stdin`.

For writing to streams there are

`fputc()` which writes one character at a time;
`fputs()` which writes one line at a time; and
`fprintf()` which writes under the control of a format string.

There are also `putchar()` and `printf()` which are the specialized versions of `fputc()` and `fprintf()` for writing to the `stdout`.

When you type a program's name, say *progname*, on the command-line at the Unix terminal and press the *Enter* key, things are arranged—not by C, but by the Unix shell—so that your keyboard's output is directed to the program's `stdin`, and the program's `stdout` and `stderr` are directed to the screen. It is important to appreciate that the program itself is not aware of this arrangement; all it "sees" are the inner ends of the streams; it has neither an awareness of, nor an interest in, whence the streams come or

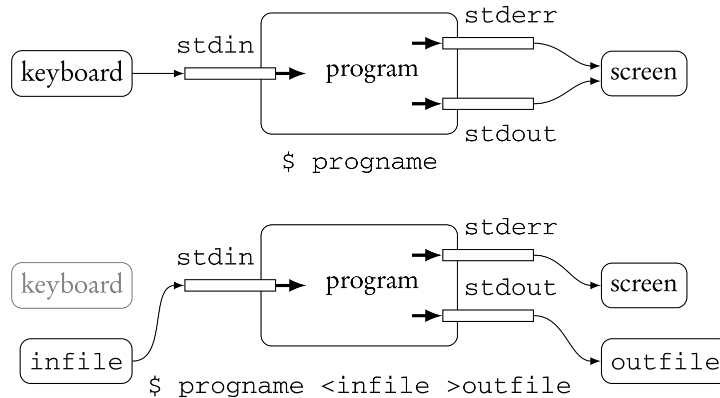


Figure 3.1: The two diagrams represent programs that read from the stdin and write to the stdout and stderr. In the top diagram the program is invoked as "progname" at the command-line in a Unix shell. (The "$" sign is the traditional symbol for the Unix shell prompt.) The shell attaches the stdin to the keyboard, and both stdout and stderr to the terminal screen. In the bottom diagram the program is invoked as "progname <infile >outfile" at the command-line, where *infile* and *outfile* are file names. The shell directs the *infile* to the program's stdin, and it directs the program's stdout to *outfile*. The stderr remains attached to the terminal screen.

whither they go. It is the Unix shell[4] that plays the "traffic cop" and directs the flow outside the program. The top half of Figure 3.1 is a cartoon depiction of this idea.

The Unix user may, and often does, alter the default flow of the traffic through the use of *pipes* and *redirection*. For instance, when you type

```
$ progname <infile >outfile
```

on the command-line, the shell directs the contents of the file *infile* to the program's stdin and directs the program's stdout to the file *outfile*. The file *infile* should exist for this to work. On the other hand, the file *outfile* need not exist—it will be created if it doesn't, and it will be overwritten if it does.[5] The stderr remains connected to the screen. The bottom half of Figure 3.1 is a cartoon illustration of this idea.

Pipes are yet another way that the Unix shell manages traffic. Let us say we have a program that generates a lot of output, say 1000 lines of text, through its stdout. We want to examine the output line-by-line rather than let it scroll away on the screen. We *pipe* the output through a *pager* such as *more* or *less*. Of these two, *more* is the older and more established but is somewhat primitive; *less* is newer and much more powerful. Almost always I use *less*.

[4]Strictly speaking, there is no such thing as *the* Unix shell. There are dozens of implementations of Unix shells each with unique characteristics and capabilities. At the time of the creation of a Unix user account, the system administrator associates a shell with the account. The Bourne shell (sh) is among the oldest of Unix shells and a standard of some sort. The modern version is bash, also known as the *Bourne-again shell*. Among the other popular shells are the *Korn shell* (ksh), the C shell (csh and its extension tcsh), the Z shell (zsh), etc. Type "echo $SHELL" (without the quotes) at the command-line to find out the name of your shell. The elementary shell constructs that I use in this book are common to *all* Unix shells; therefore the use of the phrase "the Unix shell" is not entirely misguided.

[5]To *append* to, rather than overwrite, *outfile*, we enter "$ progname <infile >>outfile". Note the doubled redirection symbol.

A pager receives a possibly very lengthy text and displays it one-screenful-at-a-time on the terminal screen. You press the space-bar to display the next screenful. Both *more* and *less* have significantly more features than this—read their man pages for the details—but this should give you an idea of what they do. Invoking the program on the Unix command-line as

```
$ progname <infile |less
```

directs *infile* to the *progname*'s `stdin` and channels *progname*'s `stdout` to *less*'s `stdin`. Did you follow that?

Finally, let us address the natural question: why are there `stdout` and `stderr`? Why does C open two output streams, and why are they treated differently by the Unix shell?

Let us say a part of a program's computational task involves solving a linear system of equations and printing the solution or, should the system be singular, printing an error message to let the user know of the issue. What is wrong in sending all the program's output to the `stdout`?

What is wrong is this: Should the user redirect the output to a file, then the error message will go into that file and possibly escape notice. On the other hand, if the error message is sent to the `stderr` while the normal output goes out on the `stdout`, the error message will appear on the screen even when the output is redirected, because as we have seen, the shell's redirection operator ">" redirects the `stdout` but not the `stderr`.

It's the programmer—that's you—who decides which parts of a program's output should be written to `stdout` and which to `stderr`. You will learn that with experience.

Chapter 4
Pointers and arrays

4.1 ▪ Pointers

Objects in C not only have values, but they also have storage locations. For instance, the declaration "**int** i;" associates a memory slot capable of holding an **int** value with the identifier i. Setting i = 3 stores a representation of the number 3 in that slot. Printing i, as in "printf("%d\n", i)", sends i's value, that is, 3, to the program's stdout.

The expression &i produces the *memory address* where i is stored. We may print that address as well:

```
printf("%p\n", (void *)&i);
```

This will send a numeral such as 0xbf84823c to the stdout.[6] The cast to "**void** *" is necessary because printf()'s %p conversion expects it. The output is likely to vary in every run of the program because there is no reason for i to be stored in the same memory location every time.

The address produced by &i is of type *pointer*, or in this specific instance, a *pointer to int*. Although a pointer appears to be of numeric sort, it definitely *is not* of type **int**. Objects of type **int** obey the regular rules of arithmetic, e.g., $1 + 12 = 13$, while objects of pointer type don't. Compare, for instance, the numbers corresponding to &i and 1 + &i:

```
printf("%p\n%p\n", (void *)&i, (void *)(1 + &i));
```

This will print something like

```
0xbf84823c
0xbf848240
```

Note that the second number is *not* 1 plus the first. In fact, it is 4 plus the first. Why 4? It's because the size (in bytes) of an object of type **int** in my computer is 4. Adding 1 to a "pointer to int" increments its value by **sizeof**(**int**). This generalizes to *the fundamental property of pointer arithmetic:*

> Adding an integer n to a "pointer to type T" increments the pointer's value by n * **sizeof**(T).

[6]The 0x prefix signals that this is a hexadecimal (i.e., base 16) representation. Hexadecimal digits are 0, 1, 2, ..., 9, a, b, c, d, e, f.

What distinguishes C from many other programming languages is its ubiquitous use of pointers and pointer arithmetic. A firm grasp of these concepts is essential for benefiting from the rest of this book.

4.2 ▪ Pointer types

Let `p = &i`, where `i` is of type **`int`**. We saw in the previous section that although on the surface the value of `p` is a number, `p` is *not* of type **`int`**. Rather, it is of type *pointer to* **`int`** since it is the *address* of an object of **`int`** type. The proper declaration of `p` is "**`int`** `*p;`". Note the asterisk.

Thus, `p` holds the memory address of where `i`'s value is stored. Since we know the address, we should be able to read its contents without referring to `i`. The expression `*p` achieves that. In general, if `p` is a pointer to an object, then `*p` is the value stored in that object.[7] The following program accesses and prints the value stored in `i`, that is, 3, in two different ways: once as `i` and once as `*p`. It also prints the address of the object's location in two different ways: once as `&i` and once as `p`. Try it out and be sure you understand every detail before moving on.

```
#include <stdio.h>
int main(void)
{
    int i = 3;
    int *p = &i;
    printf("value of i = %d, same as %d\n", i, *p);
    printf("address of i = %p, same as %p\n",
            (void *)&i, (void *)p);
    return 0;
}
```

In effect, the asterisk (`*`) and ampersand (`&`) operators are inverses of each other. One goes from value to address; the other goes from address to value. Applying the asterisk operator to a pointer is called *dereferencing the pointer*.

4.3 ▪ The pointer to void

The statement "**`void`** `*p;`" declares a special type of pointer called a *pointer to void*, or *void pointer*. There is no object of type **`void`** in C; therefore "**`void`** `*p;`" has no a priori meaning. C defines it to be a pointer of "generic type" which can point to an object of any type. This means that the usual pointer arithmetic does not apply to void pointers. What would be the meaning of `1+p`, for instance? The pointer arithmetic would want to add **`sizeof`**`(`**`void`**`)` to `p`, but since there is no object of type **`void`**, that operation would make no sense. For the same reason, it makes no sense to dereference a void pointer.

The essential property of a void pointer is that it is compatible with all other pointers. The following program illustrates this:

```
1  #include <stdio.h>
2  int main(void)
3  {
4      int a = 5;
5      double x = 3.14;
6      void *p;
7      p = &a;
```

[7] A pointer of type "void *" is an exception. More on that later.

```
8        printf("value of a = %d\n", *(int *)p);
9        p = &x;
10       printf("value of x = %g\n", *(double *)p);
11       return 0;
12   }
```

In line 7 `p` is set to hold the address of `a`, while in line 9 `p` is set to hold the address of `x`. This illustrates that a void pointer is *generic*; it is compatible with assignments to pointers to **`int`** and **`double`**, or any other type for that matter.

On line 8 we print the contents of the memory location pointed at by `p`. We noted earlier that a void pointer may not be dereferenced; therefore `*p` would be illegal. That's why we cast `p` to a pointer to **`int`** first and then dereference the result. That should explain the `*(int *)p`. The cast on line 10 serves a similar purpose.

The genericity of the void pointer enables us to write functions that may admit arguments of unspecified types. For instance,

```
int compare(void *a, void *b);
```

declares a function named `compare()` that admits a pair of pointers to objects of *any type* as arguments. That function's implementation may cast and dereference its arguments as appropriate, and return a value less than, equal to, or greater than zero, indicating that the first argument is less than, equal to, or greater than the second. Such a function is suitable for passing to the C standard library's `qsort()` function, which will sort an array according to `compare()`'s criterion. It is crucially important that the arguments of `compare()` be generic; otherwise `qsort()` would be limited to sorting arrays of only a fixed type.

4.4 ▪ Arrays

The statement "**`double`** `a[100];`" declares an *array* of length 100 of the type **`double`**. This array can hold a sequence of 100 double-precision floating point numbers. The array's ith element is `a[i]`. The array index begins with zero; therefore the array elements are `a[0]`, `a[1]`, `a[2]`, ..., `a[99]`. This is in variance with the common usage in mathematics where a vector's index begins with one; therefore some care is necessary when implementing mathematical algorithms in C. Experience shows that this cultural difference is easy to overcome. The code fragment

```
double sum = 0.0;
for (int i = 0; i < 100; i++)
    sum += a[i];
```

computes the sum of the array's 100 entries.

An array is stored in a contiguous area of the memory; that is, the addresses of the successive elements of its entries form an arithmetic progression, as illustrated in

```
#include <stdio.h>
int main(void)
{
    double a[4];
    printf("address of a[0] = %p\n", (void *)&a[0]);
    printf("address of a[1] = %p\n", (void *)&a[1]);
    printf("address of a[2] = %p\n", (void *)&a[2]);
    printf("address of a[3] = %p\n", (void *)&a[3]);
    return 0;
}
```

This program prints something like this:

```
address of a[0] = 0xbffff6d8
address of a[1] = 0xbffff6e0
address of a[2] = 0xbffff6e8
address of a[3] = 0xbffff6f0
```

Note that the addresses, represented as hexadecimal numerals, are incremented by the constant value 8 because on my machine an object of type **double** is 8 bytes wide.

Array names are handled in an interesting way in C. In most contexts—see Remark 4.1 below for an important exception—an array's name is interpreted as a pointer to the array's first element. One says that *the array's name decays to a pointer*. For instance, if you insert the extra line

```
printf("value of a = %p\n", (void *)a);
```

in the program above, you will see that it prints the same hexadecimal number that it prints for `&a[0]`.

Remark 4.1. An important exception to the decay of array names occurs in conjunction with the **sizeof** operator. In that case the array name *does not decay* to a pointer. Thus, with "`double a[100];`", the expression "`sizeof a`" evaluates to the total memory (in bytes) taken up by the entire array, that is, `100 * sizeof(double)`, which would be 800 bytes if a **double** is 8 bytes wide, as it is on my machine. Had the array decayed to a pointer, then "`sizeof a`" would have evaluated to the size of a "pointer to **double**", which would be perhaps 8 bytes—this depends on your system—and has nothing to do with the array's length.

Remark 4.2. Since "`sizeof a`" is the total memory taken up by the array, and since `sizeof a[0]` is the memory taken up by its first element (which is the same as the memory taken up by any element), then "`sizeof a / sizeof a[0]`" is the number of elements in the array. This is quite useful for finding the length, that is, the number of elements, in an explicitly initialized array, as in

```
#include <stdio.h>
int main(void)
{
    int a[] = { 1, 1, 2, 3, 5, 8, 13 };
    printf("a has %d elements\n", sizeof a / sizeof a[0]);
    return 0;
}
```

Remark 4.3. We noted earlier that in most contexts the name of the array, say `a`, is treated as the address of its first element. In view of the properties of pointer arithmetic, then `a + 1` is the address of the next element. In general, `a + i` is the address of the `i`th element. Dereferencing that, as in `*(a + i)`, yields the value stored there. But the value stored in the `i`th element is `a[i]`. We conclude that

`a + i = &a[i]` and `a[i] = *(a + i)`.

From this point of view, the notation `a[i]` is mere syntactic sugar; its true meaning is `*(a + i)`, which says begin at the address `a`, step forward `i` memory slots, and then dereference that address.

Remark 4.4. The identity `a[i] = *(a + i)` has a curious consequence:

`a[i] = *(a + i) = *(i + a) = i[a].`

Therefore, `23[a]` is a completely legal—albeit very unorthodox—way of writing `a[23]`. This is hardly more than an amusing curiosity, although some claim that this can be put to good use for writing a more expressive code in certain rare circumstances. For example, since literal strings are treated as arrays in C, both `printf()`s in the program below print the 7th letter of `"Hello World"`, that is, W:

```
#include <stdio.h>
int main(void)
{
    printf("%c\n", "Hello World"[6]);
    printf("%c\n", 6["Hello World"]);
    return 0;
}
```

You decide which is the more expressive.

Remark 4.5. Neither `*(a + i)` nor `a[i]` has built-in error-checking mechanisms. It is your responsibility as a programmer to make sure that the address `a + i` lies within the array's bounds. Stepping outside the array's bounds is a fatal error and almost certainly will crash the program with the dreaded *segmentation fault* message.

Remark 4.6. There are varying requirements on the declaration of arrays such as "`int a[n];`" in C89, C99, and C11. In C89, the value of `n` is required to be a numeric constant which is known at the time when the program is compiled. C99 allows the value of `n` to be determined during the program's execution. Such so-called *variable-length arrays* proved to be quite unpopular when C99 was announced and raised strong objections within the C programming community. C11 strikes a compromise by making the support for variable-length arrays optional. This in effect eliminates the use of variable-length arrays in writing portable programs because their support cannot be guaranteed on all compilers.

For this reason I will refrain from using C99's variable-length arrays in this book. *Dynamically allocated arrays*, the topic of Chapter 8, describe the portable way of dealing with arrays whose sizes are not predetermined.

4.5 ▪ Multidimensional arrays

The declaration

```
double a[3][4];
```

makes a two-dimensional array that may be used to store a matrix of 3 rows and 4 columns. The element in row i and column j is accessed through the syntax `a[i][j]`. The expression `a[0][2] = 3.14` assigns the value 3.14 to the $(0, 2)$ element. Higher-dimensional arrays work in the same way. For instance, the elasticity tensor of a linearly elastic material is a $3 \times 3 \times 3 \times 3$ array which may be declared as

```
double C[3][3][3][3];
```

Its elements will be accessed through the syntax `C[i][j][p][q]`.

4.6 ▪ Strings

There is no intrinsic string data type in C. A string is defined to be a contiguous (in memory) sequence of characters terminated by and including the ASCII `NUL` character, `\0`, which is called the string's *null terminator*. For instance, the array `s[]` defined by

```
char s[8];
s[0] = 'T'; s[1] = 'e'; s[2] = 'x'; s[3] = 't'; s[4] = '\0';
```

represents the string "Text". The string's length is 4, but it takes up 5 bytes of storage because of the inclusion of the `NUL`. The array's elements 5, 6, and 7 are past the null terminator; therefore they are not a part of the string. Moreover, since they have not been assigned to, they will hold garbage values. When we pass a string to a function, we need not supply the string's length because the receiving function can tell the string's end by its null terminator.[8]

The length and contents of the string defined above may be changed as long as we don't exceed the array bounds. For instance,

```
s[2] = 's'; s[4] = 'i'; s[5] = 'n'; s[6] = 'g';  s[7] = '\0';
```

changes the string from "Text" to "Testing".

The declaration and initialization steps of a string may be merged into a single statement, as in

```
char s1[8] = { 'T', 'e', 'x', 't', '\0' };
```

or simply:

```
char s2[] = { 'T', 'e', 'x', 't', '\0' };
```

In the latter case, the length of the array will be exactly 5.

It is also possible to define a string simply as

```
char s3[] = "Text";
```

or

```
char *s4 = "Text";
```

Any of the strings `s1` ⋯ `s4` may be used with identical results in most contexts where strings are used—all four represent "Text". However, there are subtle differences that one needs to be aware of, especially in relation to the **`sizeof`** operator, as can be seen here:

```
#include <stdio.h>
int main(void)
{
    char s1[8] = { 'T', 'e', 'x', 't', '\0' };
    char s2[]  = { 'T', 'e', 'x', 't', '\0' };
    char s3[]  = "Text";
    char *s4   = "Text";

    printf("s1 = %s, sizeof s1 = %d\n", s1, sizeof s1);
    printf("s2 = %s, sizeof s2 = %d\n", s2, sizeof s2);
    printf("s3 = %s, sizeof s3 = %d\n", s3, sizeof s3);
    printf("s4 = %s, sizeof s4 = %d\n", s4, sizeof s4);
    return 0;
}
```

[8] Old joke floating on the Internet: These two strings walk into a bar and sit down. The bartender says, "So what'll it be?" The first string says, "I think I'll have a beer fulk boorg jdk@¿ÇjfdŁ¡ kjk3s d#f67hower89åøvyo… "Please excuse my friend," the second string says, "she isn't null-terminated."

This prints

```
s1 = Text, sizeof s1 = 8
s2 = Text, sizeof s2 = 5
s3 = Text, sizeof s3 = 5
s4 = Text, sizeof s4 = 4
```

Can you explain the result? Recall the discussion of the decay of array names and the exception noted in Remark 4.1 on page 22.

Remark 4.7. A quoted set of characters, such as the `"Text"` in the definition of `s3` and `s4` above, is called a *string literal*. Your C compiler is permitted to treat a string literal as a read-only object, making it impossible to change its letters. For instance, with the `s1` and `s4` defined as above, setting `s1[0] = 'N'` is quite legal and changes `s1` to "Next", but setting `s4[0] = 'N'` is illegal and may have unpredictable results.

The C standard library provides a dozen or so functions for working with strings. There are functions to copy, concatenate, search, compare, tokenize, and print strings. For instance the `strlen()` function gives a string's length (not counting the null terminator.)

4.7 ▪ The command-line arguments

Programs are often invoked with command-line arguments and options. For instance, the Unix utility `diff` takes two file names as arguments, as in `diff file1 file2`, and prints the differences between the contents of the two files. How are the file names `file1` and `file2` conveyed from the command-line to the `diff` program?

C provides two alternative forms for declaring the function `main()`. If the program has no use for command-line arguments, then we use the simpler alternative, "`int main(void)`". If the program needs to access the command-line arguments, then we use the second alternative: "`int main(int argc, char **argv)`". In the latter case, the `argc` parameter will hold the number of command-line tokens with which the program is invoked. For instance, if the command-line is "`diff file1 file2`", then the value of `argc` will be set to 3. If a program is invoked with no arguments, then the only command-line token is the program's name; therefore `argc` is set to 1.

The `argv` parameter is an array of length `argc + 1` of pointers to **`char`**; `argv[0]` points to a string corresponding to the first command-line token, which is the name of the program; `argv[1]` through `argv[argc-1]` point to strings corresponding to the remaining command-line tokens; `argv[argc]` is the `NULL` pointer—it marks the end of the `argv` array much in the same way that the null terminator marks the end of a string. Figure 4.1 is a schematic representation of the idea.

The following program prints the command-line arguments and exits:

```
#include <stdio.h>
int main(int argc, char **argv)
{
    while (*argv ≠ NULL)
        printf("%s\n", *argv++);
    return 0;
}
```

Can you think of other ways of doing this?

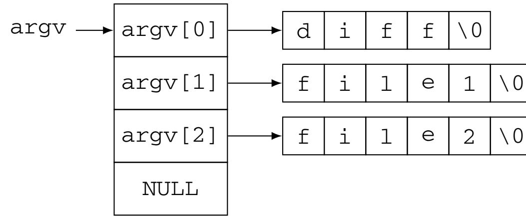


Figure 4.1: The schematic representation of the memory layout of the command-line arguments when the program is invoked as "`diff file1 file2`".

Remark 4.8. This section's discourse on the values of `argc` and `argv` is intentionally simplistic and not quite accurate. My aim has been to cover the most common case, and in that sense what I have written *is* accurate. It is possible, however, to step outside of the ordinary and encounter behaviors which are quite at odds with what I have said. For instance, Unix's `exec()` family of functions allows one to execute a program in such a way that `argc` appears to be zero or in such way that `argc` is nonzero but `argv[0]` is something other than the program's name. These issues are not of relevance to this book's material; therefore I will not go into them.

Chapter 5

From strings to numbers

In Section 4.7 we saw how to access the user-supplied command-line arguments as strings `argv[0]`, `argv[1]`, etc. If any of these arguments is supposed to be interpreted as a number, then we need a means of converting a string to a number. The C standard library provides several functions for that purpose. In this chapter we will learn about a few of these which are used frequently in the rest of this book.

5.1 ▪ The function `strtod()`

The standard library's `strtod()` (think of it as *string to double*) function "converts the *initial portion* of a string" to a number of type **`double`**. The expression in the quotes, which comes from the C standard, requires some explanation. The function is declared in the standard header file *stdlib.h* as

```
double strtod(const char *nptr, char **endptr);
```

We see that it takes two arguments. The first argument is expected to be a pointer to a string that expresses a floating point number such as "`3.14`", "`-0.123`", or "`6.0221415e+23`". Leading whitespace, if any, is ignored. The parsing of the string stops when the end of the number is detected. Thus, the string may include trailing junk. For instance, "`␣␣␣6.022e+23xyz`" yields the same number as "`6.022e+23`". Here I have used "␣" to mark whitespace characters.

The second argument, called the *end-pointer*, is the address of a pointer to **`char`**. Upon reaching the end of the numeric portion of the string, `strtod()` sets the end-pointer to point to the memory location next to the number's last character. For instance, if the string is "`6.022e+23xyz`", then the end-pointer points to the character "x". In particular, if there is no trailing junk, then the end-pointer points to the string's null terminator, that is, the "`\0`" character. It follows that if, after `strtod()` returns, the end-pointer is pointing to anything other than "`\0`", then the string contains trailing junk. It's conceivable that the user put the junk there on purpose, but more often than not, trailing junk is the result of an unintended error in the input—perhaps a slip of the finger on the keyboard—and we would want to catch that. Therefore, a typical call to `strtod()` takes the form

```
#include <stdlib.h>
char *str = ...; // some string here
char *endptr;
```

```
double x;
x = strtod(str, &endptr);
if (*endptr ≠ '\0')
    fprintf(stderr, "your string has trailing junk!\n");
else
    printf("the number is %g\n", x);
```

Remark 5.1. The standard library also provides `strtof()` and `strtold()`, which return floating point numbers of types **`float`** and **`long double`**, respectively. The function `strtod()` dates back to C89. The other two were added in C99. We will have use only for `strtod()` in this book.

5.2 ▪ The function `strtol()`

The standard library's `strtol()` (think of it as *string to long int*) function "converts the *initial portion* of a string" to a number of type **`long int`**. The expression in the quotes comes from the C standard which, in the header file *stdlib.h*, declares the prototype:

```
long int strtol(const char *nptr, char **endptr, int base);
```

We see that `strtol()` takes three arguments. The first argument is expected to be a string that expresses an integer such as "`123`", "`-321`", or "`beef`" (yes, that's a hexadecimal number). Initial whitespace, if any, is ignored. The parsing of the string stops when the end of the number is detected. Thus, the string may include trailing junk. For instance, "`␣␣␣-321#@x`" yields the same number as "`-321`".

The second argument, called the *end-pointer*, works exactly like that in the case of `strtod()` described in the previous section; therefore I will not repeat that description here. The third argument is the *base* in which the number is to be interpreted. For decimal numbers the base is 10. For hexadecimal numbers it is 16. The base may be as small as 2 (a binary number) or as large as 35. The digits in base 35 are $0, 1, \dots, 9, a, b, \dots, z$ (uppercase or lowercase letters are equivalent). Setting the base to zero has a special meaning which is not worth getting into in this brief overview. Read your C reference to find out if you are curious.

Here is a typical call to `strtol()`:

```
#include <stdlib.h>
char *str = ...; // some string here
char *endptr;
long int n;
n = strtol(str, &endptr, 10);
if (*endptr ≠ '\0')
    fprintf(stderr, "your string has trailing junk!\n");
else
    printf("the number is %d\n", n);
```

Remark 5.2. The standard library also provides the function `strtoll()` which returns extended range integers of the type **`long long int`**. You may use `strtoll()` to find out the base 10 equivalent of the hexadecimal number `deadbeef`. On a typical computer it is likely that `deadbeef` is beyond the range of **`long int`**, so `strtol()` won't do.

There are also `strtoul()` and `strtoull()` which return **`unsigned long int`** and **`unsigned long long int`**. The functions `strtol()` and `strtoul()` date

back to C89. The other two were added in C99. We will have use only for `strtol()` in this book.

5.3 ▪ The functions atof(), atol(), and friends

The C standard library function `atof()` (think of it as *ascii to float*) is the lazy man's version of `strtod()`. It is declared in *stdlib.h* as

```
double atof(const char *nptr);
```

Thus, `atof()` takes a pointer to a string and produces a floating point number of type **double**. Initial whitespace and trailing junk, if any, *are silently ignored*. Therefore, `atof(␣␣␣6.022e+23xyz)` returns `6.022e+23`.

The C standard library function `atol()` (think of it as *ascii to long int*) is the lazy man's version of `strtol()`. It is declared in *stdlib.h* as

```
long int atol(const char *nptr);
```

Thus, it takes a pointer to a string and produces an integer of type **long int**. Initial whitespace and trailing junk, if any, *are silently ignored*. Therefore, `atol(␣␣␣-321xyz)` returns `-321`. Unlike `strtol()`, there is no provision for bases other than 10.

There are also `atoi()` and `atoll()` which work just like `atol()` but return **int** and **long long int**, respectively. The functions `atof()`, `atol()`, and `atoi()` date back to C89. The function `atoll()` was introduced in C99.

This section's `ato*()` functions can be handy for quick-and-dirty jobs and throw-away programs. For a work of any permanence, I recommend `strtod()` and `strtol()`. We won't use the `ato*()` functions in this book.

Chapter 6

Make

6.1 ▪ Multifile programs

It helps to split a program into as many logically independent parts as possible. Generally a small programming unit is easier to comprehend, debug, and improve. A "logically independent part" does not stand in vacuum, however. It is always connected through thin filaments to the other parts; otherwise one could remove it and nothing would be the worse for it.

The "parts" of the program generally manifest themselves as files or groups of files. Programs that are spread over dozens, if not hundreds, of files are quite common. Unix's *make* utility is an essential tool for managing a multipart program.

To provide a context to this exposition, I will refer to Chapter 17's *Project Evolution*, although a knowledge of the details of that chapter is not a prerequisite for understanding this chapter. *Project Evolution* is spread over five **.c* files and six **.h* files:[9]

```
$ cd evolution
$ ls -F
array.h@     read.c          world-to-eps.h  xmalloc.c@
evolution.c  read.h          write.c         xmalloc.h@
evolution.h  world-to-eps.c  write.h
```

It's possible to compile the entire program via a single command:[10]

```
$ gcc -Wall -pedantic -std=c99 -O2 evolution.c  read.c \
    world-to-eps.c  write.c  xmalloc.c -o evolution
```

Indeed, this will compile the five **.c* files and produce an executable file named "*evolution*"; the `-o` flag precedes the name of the desired executable. The **.h* files are not listed explicitly; the compiler inserts them where indicated by the preprocessor's **`#include`** directives.

6.2 ▪ Separate compilation and linking

The single-line compilation method works, but it's rather primitive and is hardly ever advisable. If you edit one of the files, say *write.c*, then of course you will have to recompile it,

[9]Files decorated with "`@`" are *symbolic links*, as explained in Chapter 2.

[10]I have broken the line in two, as it is too long to fit between the margins of this book. A backslash (\) at the end of a Unix command-line indicates that the line is continued into the next. See Section 1.5 regarding the compilation flags `-Wall -pedantic -std=c99 -O2`.

but there should be no need to recompile the rest of the **.c* files if you haven't touched them. The sensible thing to do is to compile the files individually, and when a particular file changes, then recompile just that one, leaving the rest alone. Here is how that's done:

```
$ gcc -c -Wall -pedantic -std=c99 -O2 evolution.c
$ gcc -c -Wall -pedantic -std=c99 -O2 read.c
$ gcc -c -Wall -pedantic -std=c99 -O2 world-to-eps.c
$ gcc -c -Wall -pedantic -std=c99 -O2 write.c
$ gcc -c -Wall -pedantic -std=c99 -O2 xmalloc.c
```

The `-c` flag tells `gcc` to compile each **.c* file into a corresponding **.o* file, called an *object file*. Although an object file is a finished product as far as compilation goes, it's not suitable for stand-alone execution since it's only a fragment of a whole program. Consider, for instance, the first of the five lines shown above. Within *evolution.c* there is a reference to the function `read_world_def()` which is defined elsewhere (in *read.c*, to be exact). The header file *read.h*, which is **`#include`**`d` in *evolution.c*, reassures the compiler that the function `read_world_def()` with a specific prototype will be made available later. That satisfies the compiler, and it produces the object file *evolution.o* with a dangling reference to the promised `read_world_def()`.

There are many more dangling references in *evolution.o*. In *evolution.c* there are several references to the function `printf()`, for instance, which is defined elsewhere (in the C standard library, to be exact). The header file *stdio.h*, which is **`#include`**`d` in *evolution.c*, reassures the compiler that the function `printf()` with a specific prototype will be made available later.

The loose ends are tied together in a process called *linking*. It's done by giving the names of the object files to `gcc`, as in

```
$ gcc evolution.o read.o world-to-eps.o write.o xmalloc.o \
        -o evolution
```

Again I have broken the long line in two, but you can see what is going on. There are no compilation flags, such as `-Wall`, because no compilation takes place here. `Gcc` sees a list of object files and recognizes the request for linking. It ties the loose ends among the listed **.o* files and the standard library, and produces the executable file named "*evolution*" because that's what the `-o` flag tells it.

Suppose we make a change in *write.c*. Producing a new executable amounts to

```
$ gcc -c -Wall -pedantic -std=c99 -O2 write.c
$ gcc evolution.o read.o world-to-eps.o write.o xmalloc.o \
        -o evolution
```

That is, we recompile *write.c* and relink. The other **.c* files were not touched; therefore they need not be recompiled.

6.3 ▪ File dependencies

This is all quite straightforward until we change a header file, say *write.h*. Which files need to be recompiled now? We need to inspect each of the other **.c* and **.h* files to find out which of them **`#include`**`s` the file *write.h* and then recompile the affected files. Doing this manually can be daunting because a program's various files may be interrelated in complex ways. Figure 6.1 shows the dependencies of the files in *Project Evolution*. At the very top is the executable *evolution* which depends on the five **.o* files. We see that the object file *write.o* depends on *write.h*, *write.c*, and *evolution.h*. We also see that the object file *evolution.o* depends on *evolution.c* as well as the six header files **.h*.

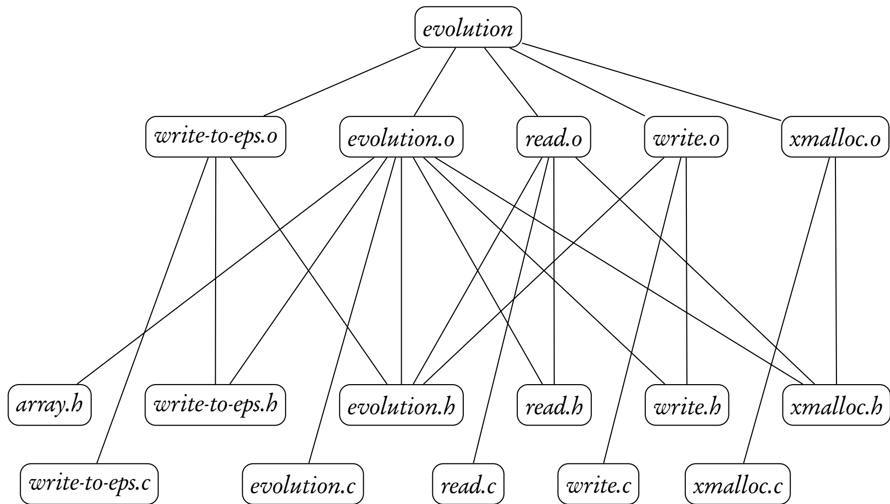


Figure 6.1: This diagram shows the dependency tree among the files pertaining to *Project Evolution*. The executable *evolution* (at top) depends on the **.o* files in the second row. Each of the **.o* files depends on one or more **.c* and **.h* files, listed at the bottom two rows.

Fortunately, we don't have to discover these dependencies through laborious manual inspection; `gcc`'s `-MM` flag will do that for us. In response to the command

```
$ gcc -MM evolution.c read.c world-to-eps.c write.c xmalloc.c
```

it prints

```
evolution.o: evolution.c evolution.h xmalloc.h array.h read.h \
        write.h world-to-eps.h
read.o: read.c xmalloc.h read.h evolution.h
world-to-eps.o: world-to-eps.c world-to-eps.h evolution.h
write.o: write.c write.h evolution.h
xmalloc.o: xmalloc.c xmalloc.h
```

Each line has the form "target: dependency$_1$ dependency$_2$ ...". For instance, we see that *write.o* depends on *write.c*, *write.h*, and *evolution.h*, as was noted earlier. Item or items to the left of the colon are called *targets*. Item or items to the right of the colon are called *dependencies*.

The form of the output of `gcc -MM` is designed specifically to mesh with the Unix utility called `make`. `Make` reads a file, usually called *Makefile*, that contains the list of targets/dependencies of a project in the textual form shown above. It then figures out what needs to be done to compile the program without further human intervention.

You, the programmer, write the appropriate *Makefile* for your program. Writing a *Makefile* may seem akin to black magic if you don't know the principles behind it. In the next several sections I will construct a sequence of *Makefile*s, initially some rather primitive ones and then progressively more sophisticated and complex ones but with greater utility. The later versions are much more practical, and I recommend that you use those rather than the simplistic early versions, but *don't skip over the earlier versions* as you read this chapter; they show the progression of logic that leads to the somewhat complex final product which may not be easily decipherable otherwise.

Listing 6.1: *Makefile* version 1. Make recognizes a backslash \ at the end of a line as a directive to continue to the next line, as the Unix shell does.

```
1  evolution: evolution.o read.o world-to-eps.o write.o xmalloc.o
2          gcc evolution.o read.o world-to-eps.o write.o xmalloc.o \
3                          -o evolution
4
5  evolution.o: evolution.c evolution.h xmalloc.h array.h read.h \
6                          write.h world-to-eps.h
7          gcc -Wall -pedantic -std=c99 -O2 evolution.c
8
9  read.o: read.c xmalloc.h read.h evolution.h
10         gcc -Wall -pedantic -std=c99 -O2 read.c
11
12 world-to-eps.o: world-to-eps.c world-to-eps.h evolution.h
13         gcc -Wall -pedantic -std=c99 -O2 world-to-eps.c
14
15 write.o: write.c write.h evolution.h
16         gcc -Wall -pedantic -std=c99 -O2 write.c
17
18 xmalloc.o: xmalloc.c xmalloc.h
19         gcc -Wall -pedantic -std=c99 -O2 xmalloc.c
```

6.4 ▪ *Makefile* version 1

The first version of our *Makefile* shown in Listing 6.1 is quite verbose but also quite simple to comprehend. Line 1 says that the target file called "*evolution*" depends on the five **.o* files shown. That single line is exactly equivalent to the top half of Figure 6.1.

As it stands, line 1 is incomplete: it asserts *what* the target depends on but does not tell *how to build it* from its dependencies. We, the humans, know that it's done through linking:

```
$ gcc evolution.o read.o world-to-eps.o write.o xmalloc.o \
        -o evolution
```

but make doesn't know that and needs to be told. That's where line 2 comes in. In a *Makefile* we may associate a *command* with a target/dependency pair. The command tells make what needs to be done to produce the target out of the dependencies. The syntax is

target	:	dependencies
← tab →		command

The *tab* that precedes the command is a literal *Tab* character and is *absolutely essential*; spaces won't do. The GNU make's manual thus puts it emphatically: "You need to put a tab character at the beginning of every command line! This is an obscurity that catches the unwary."

Equipped with this knowledge, you should have no problems in deciphering the *Makefile* of Listing 6.1. Line 15, for instance, says that *write.o* depends on *write.c*, *write.h*, and *evolution.h*. Line 16 supplies the command that produces *write.o* out of those dependencies. The indentation of that line is due to the presence of the leading tab.

6.5 ▪ How to run `make`

At this point we have a file named *Makefile* whose contents are shown in Listing 6.1. Execute the Unix command

```
$ make
```

and—*voilà!*—`make` reads and executes the instructions in the *Makefile* and produces the file *evolution*. What is even more impressive, if you change any of your source files and run `make` again, *it will detect what has changed* and will know exactly which files to recompile—since it knows their interdependencies—to create a new *evolution* file. If you rerun `make` without having made any changes, it will know that there is nothing to do and will tell you so.

Remark 6.1. Occasionally it is useful to run `make` with the `-n` flag:

```
$ make -n
```

This does a "dry run"—it prints out what will be done without doing them.

Let us look once more at the *Makefile* in Listing 6.1. Note that there are six targets in it. When we run `make`, it picks up *the very first target* to build. It sees that the first target, that is, *evolution*, depends on *evolution.o* among other things. So it looks up and executes the instructions for building *evolution.o*. It repeats this for the other **.o* files and then puts them together to build the first target.

In view of the special role of a *Makefile*'s first target, it is a good idea to put the most frequently needed object as the first target in a *Makefile*. This is not absolutely necessary, however. Had we placed the *evolution* target somewhere else, let's say at the end of the *Makefile*, we still could have requested `make` to build it by giving the name of the target explicitly on the command-line:

```
$ make evolution
```

Putting the most frequently needed object as the first target in a *Makefile* makes life a little easier since it won't be necessary to type the target's name on the command-line.

6.6 ▪ *Makefile* version 2

Line 16 of Listing 6.1 instructs `make` to compile *write.c* in order to build *write.o*. But that's rather obvious; how else would one build *write.o* out of *write.c*? Can't `make` figure that out by itself? It turns out that yes, it can! `Make` has a built-in rule for producing an object file from a C file. If we don't specify the command ourselves, `make` will apply the built-in rule. This means that line 16 is superfluous and may be removed.

There remains an issue regarding `gcc`'s compilation options. If we remove line 16, then how do we tell `make` to compile *write.c* with our favorite compilation flags?

`Make` recognizes a built-in variable named `CFLAGS`. We may define `CFLAGS` in the *Makefile* as we wish. `Make` will apply `CFLAGS` along with its built-in compilation rules. Implementing these observations, we arrive at version 2 of our *Makefile* shown in Listing 6.2. Relative to the previous version, I have removed the commands associated with the **.o* targets to let `make` apply its own built-in rules instead. Note the `CFLAGS` defined at the top.

Lines 3 and 4 have remained exactly as before because we still need to tell `make` how to build the *evolution* target from the object files. That part is very much project-dependent; no built-in rule can help there.

Listing 6.2: *Makefile* version 2.

```
1  CFLAGS = -Wall -pedantic -std=c99 -O2
2
3  evolution: evolution.o read.o world-to-eps.o write.o xmalloc.o
4          gcc evolution.o read.o world-to-eps.o write.o xmalloc.o \
5                          -o evolution
6
7  evolution.o: evolution.c evolution.h xmalloc.h array.h read.h \
8                          write.h world-to-eps.h
9  read.o: read.c xmalloc.h read.h evolution.h
10 world-to-eps.o: world-to-eps.c world-to-eps.h evolution.h
11 write.o: write.c write.h evolution.h
12 xmalloc.o: xmalloc.c xmalloc.h
```

Remark 6.2. Lines from 7 on down are *exactly* what "`gcc -MM`" produces (see the output shown on page 33); therefore one does not type those lines by hand. I certainly didn't.

There are several ways to grab the output of "`gcc -MM`" and stick it into the *Makefile*. If you are working on a terminal that allows copying and pasting with the mouse, you may do just that, that is, copy the output of "`gcc -MM`" and paste it into the *Makefile*, but that is probably the clumsiest way of doing it. A better way is to rely on your Unix shell's redirection ability to append[11] `gcc`'s output to the *Makefile*:

```
$ gcc -MM evolution.c read.c world-to-eps.c \
            write.c xmalloc.c >>Makefile
```

You should remember, however, as you add and remove files, or change any **`#include`** lines in your sources, or anything else that may alter the file dependency tree, you need to refresh those dependency lines. To do that, edit the *Makefile*, delete the previous dependency lines, and then regenerate them as instructed above.

There are ways to automate the updating of the dependency lines in a *Makefile* but they will probably be more confusing to a beginner than of help, so I won't go there.

6.7 ▪ *Makefile* version 3

There is something disconcerting about version 2 of the *Makefile* in Listing 6.2: line 4 repeats the file names from line 3. That's rather annoying because any change in one line requires the identical change in the other. To avoid repetition, we introduce a new variable, let's call it `OFILES`,[12] that declares the names of the object files:

```
OFILES = evolution.o read.o world-to-eps.o write.o xmalloc.o
```

Then we may refer to that variable instead of listing the object files explicitly. To refer to `OFILES`, we write `$(OFILES)`. Thus, lines 3 and 4 simplify to

```
evolution: $(OFILES)
        gcc $(OFILES) -o evolution
```

Can it be made any simpler? Yes, it can! The word "`evolution`" appears in two places. We may avoid the repetition by changing the above to

[11] See the remark in footnote 5 on page 16.

[12] Unlike `CFLAGS`, which is a `make` built-in, the name "`OFILES`" has no special significance; you may call it as you wish. Many programmers use "`OBJS`" for this purpose.

```
evolution: $(OFILES)
        gcc $(OFILES) -o $@
```

The "`$@`" is an internal variable in `make` that holds the name of the associated target. The choice of the symbol "`@`" is apropos since it looks so much like a target/bull's-eye.

Let us remark that we have managed to reduce the bulky first three lines of *Makefile* version 1 (see Listing 6.1 on page 34) to a quite simple and almost generic two-liner. It would be even more generic if could we get rid of the occurrence of the word "*evolution*" there as well. In fact, we can. Let us define a new variable, let's call it `TARGET`, as in

```
TARGET = evolution
```

Then the previous block of code changes to

```
$(TARGET): $(OFILES)
        gcc $(OFILES) -o $@
```

To make this completely generic, we change `gcc` to `$(CC)`. The latter is defined to expand to the name of your system's default C compiler. On most Linux machines `$(CC)` expands to `cc`, which is a link to `gcc`. If your system is equipped with multiple C compilers, then you may define `CC` in your *Makefile*, as in

```
CC = icc
```

which tells `make` to use *Intel*'s C compiler, `icc`, instead of the default.[13] With this final change, we arrive at

```
$(TARGET): $(OFILES)
        $(CC) $(OFILES) -o $@
```

Hooray! This is completely generic! There are no references there to any specific project or any compiler; therefore this fragment of the *Makefile* will be the same in all projects. We celebrate this achievement by making version 3 of our *Makefile*, shown in Listing 6.3. In effect, only the first two lines in that *Makefile* are project-dependent. Everything else is generic. I am not counting the material that appears below line 8 since that is automatically generated by "`gcc -MM`".

Remark 6.3. The "**#**" character in a *Makefile* signals the beginning of a comment. `Make` ignores everything from there to the end of line. The entire line 8 of Listing 6.3 is a comment.

Remark 6.4. Up to this point I have adhered strictly to the syntax and semantics of original `make`, which dates back to 1977. You should expect that any implementation of the `make` utility—and there are quite a few—should respect the *Makefile*s constructed up to this point, and should process them as intended.

The more recent versions of `make` add features of their own as extensions to the original `make`. One of the more popular implementations, the GNU `make`, adds a "`$^`" internal variable which expands to the list of a target's dependencies. Thus, the second occurrence of `$(OFILES)` in the previous code snippet may be replaced by "`$^`", as in

```
$(TARGET): $(OFILES)
        $(CC) $^ -o $@
```

[13] Compiler flags vary wildly from one compiler to another. If you switch to another compiler, you may have to change `CFLAGS` accordingly.

Listing 6.3: *Makefile* version 3.

```
1  OFILES = evolution.o read.o world-to-eps.o write.o xmalloc.o
2  TARGET = evolution
3  CC = gcc
4  CFLAGS = -Wall -pedantic -std=c99 -O2
5  $(TARGET): $(OFILES)
6          $(CC) $(OFILES) -o $@
7
8  # below this is the output of "gcc -MM":
9  evolution.o: evolution.c evolution.h xmalloc.h array.h read.h \
10                          write.h world-to-eps.h
11 read.o: read.c xmalloc.h read.h evolution.h
12 world-to-eps.o: world-to-eps.c world-to-eps.h evolution.h
13 write.o: write.c write.h evolution.h
14 xmalloc.o: xmalloc.c xmalloc.h
```

There is not much of an advantage in using "`$^`" in this particular instance, but it can be quite handy in a *Makefile* that builds multiple executables, e.g.,

```
target1: file1.o file2.o
        $(CC) $^ -o $@

target2: file3.o file4.o
        $(CC) $^ -o $@
```

You will have to balance the convenience of using "`$^`" against the possible lack of portability.

6.8 ▪ *Makefile*: The final version

The program for *Project Evolution* consists of five **.c* files and six **.h* files. Compiling the program adds five **.o* files as well as the executable which we have named *evolution*. Occasionally you will want to "clean up" the directory by removing all machine-generated files, leaving only the original source files. You want to do that when sending your program to someone else—there is no point in sending object files and executables since these are specific to your operating system. Such a cleanup is also helpful when you wish to recompile the entire program from scratch and watch for any compiler warnings that you may have missed earlier.

We may remove the object file and executables in *Project Evolution* through the Unix command-line

```
$ rm evolution.o read.o world-to-eps.o write.o \
    xmalloc.o evolution
```

but that's too tedious and error-prone, especially if we have a large number of files. We may enlist `make` to help us here as well. Note that the parameters given to the "`rm ...`" command above are exactly what were called `$(OFILES)` and `$(TARGET)` in our *Makefile*. Therefore, within the *Makefile*, that command takes the form

```
rm $(OFILES) $(TARGET)
```

Listing 6.4: *Makefile*, final version. Only the first two lines are project-dependent. Everything else is generic. I am not counting the material that appears toward the end because that is automatically generated by "`gcc -MM`".

```
1  OFILES = evolution.o read.o world-to-eps.o write.o xmalloc.o
2  TARGET = evolution
3  CC = gcc
4  CFLAGS = -Wall -pedantic -std=c99 -O2
5  $(TARGET): $(OFILES)
6          $(CC) $(OFILES) -o $@
7
8  clean:
9          rm -f $(OFILES) $(TARGET)
10
11 # below this is the output of "gcc -MM":
12 evolution.o: evolution.c evolution.h xmalloc.h array.h read.h \
13                         write.h world-to-eps.h
14 read.o: read.c xmalloc.h read.h evolution.h
15 world-to-eps.o: world-to-eps.c world-to-eps.h evolution.h
16 write.o: write.c write.h evolution.h
17 xmalloc.o: xmalloc.c xmalloc.h
```

Where do we put that line in the *Makefile*? We make up a "phony" target, let's call it "clean", which has no dependencies and whose associated command performs the removal:

```
clean:
        rm $(OFILES) $(TARGET)
```

As we noted in Section 6.5, typing "`make`" on the command-line with no arguments picks up the very first target in the *Makefile*. To pick up some other target, we give it as an argument to `make`. Thus, to reach the "clean" target, on the Unix command-line we type

```
$ make clean
```

This will work fine, but it can be improved. It's true that the command "`make clean`" will remove `$(OFILES)` and `$(TARGET)`, as intended, but executing the command "`make clean`" for a second time will elicit a complaint from `rm` since there will be no such files to remove. The `-f` flag suppresses such complaints from `rm`:

```
clean:
        rm -f $(OFILES) $(TARGET)
```

Remark 6.5. Our use of the target "clean" constitutes an abuse of the `make` utility. `Make` is designed with the assumption that targets will be actual file names, but "clean" is not a file name; it's a phony target. This is not an issue unless for some reason you happen to have a file named "clean" in the same directory. That will confuse `make` to no end. The generic solution is to avoid having a file named "clean", or if you must, then consider changing the name of the target to something else, like "cleanup". A solution specific to GNU `make` is to explicitly declare `clean` a "phony target". Look up the Phony Targets section in GNU `make`'s manual if you are interested.

We add the "clean" target to our *Makefile* and arrive at the final version shown in Listing 6.4. Admittedly, calling it the "final version" is somewhat premature: it is missing

important features such as a means to link with external libraries. Nevertheless, what we have here is a good foundation to build upon, as we will see in the following sections.

6.9 ▪ Linking with external libraries

We will have several occasions in this book to link our programs with third-party libraries. For instance, the finite element solver of Chapter 25 relies on the *Triangle* library to triangulate the domain, the UMFPACK library to solve the resulting linear system, and the standard library's mathematics library for mathematical functions, such as `sqrt()`. To link with these libraries, we pass the `-ltriangle -lumfpack -lm` flags to the linker. A good way of doing this in a *Makefile* is to define

```
LIBS = -ltriangle -lumfpack -lm
```

and then change the command that performs the linking to

```
$(TARGET): $(OFILES)
	$(CC) $(OFILES) $(LIBS) -o $@
```

6.10 ▪ Multiple executables in one *Makefile*

In *Project Nelder–Mead* of Chapter 18 we write several independent demonstration programs named *demo-1D*, *demo-2D*, *demo-energy*, *demo-constrained*, etc. Here are the rules to build the first two:

```
OFILES_1D = xmalloc.o nelder-mead.o demo-1D.o
OFILES_2D = xmalloc.o nelder-mead.o demo-2D.o

demo-1D: $(OFILES_1D)
	$(CC) $(OFILES_1D) -o $@

demo-2D: $(OFILES_2D)
	$(CC) $(OFILES_2D) -o $@
```

If *demo-1D* is the first target, then the command `make`, with no arguments, will build the executable *demo-1D*. To produce the remaining executables, we will have to give their names on the command-line, as in "`make demo-2D`", "`make demo-energy`", "`make demo-constrained`", etc. This can be irksome if we have to rebuild the targets repeatedly. Fortunately, a simple trick can automate the building of any number of targets with a single command. Listing 6.5 shows how.

There we define the variable `ALL` as the list of the desired executables. As the *Makefile*'s first target we define a phony target called `all` which depends on `$(ALL)`. Thus, entering `make` on the command-line picks up that first target, that is, `all`. Since `all` depends on the list of executables, building `all` amounts to building all the executables.

Listing 6.5: This is a typical *Makefile* for building multiple executables. Entering the command `make` on the command-line will pick the first target, `all`, which depends on `$(ALL)`, which in turns is the list of the desired executables.
Note: If you are willing to use GNU `make`'s `$^` extension (see Remark 6.4), then you may simplify this *Makefile* by replacing every "`$(CC) ...`" line with "`$(CC) $^ -o $@`".

```
OFILES_RANK_VERTICES = rank-vertices-test.o
OFILES_1D = xmalloc.o nelder-mead.o demo-1D.o
OFILES_2D = xmalloc.o nelder-mead.o demo-2D.o
OFILES_ENERGY = xmalloc.o nelder-mead.o demo-energy.o
OFILES_CONSTRAINED = xmalloc.o nelder-mead.o demo-constrained.o

CFLAGS = -Wall -pedantic -std=c99 -O2

ALL = rank-vertices-test demo-1D demo-2D demo-energy demo-constrained

all: $(ALL)

rank-vertices-test: $(OFILES_RANK_VERTICES)
	$(CC) $(OFILES_RANK_VERTICES) -o $@

demo-1D: $(OFILES_1D)
	$(CC) $(OFILES_1D) -o $@

demo-2D: $(OFILES_2D)
	$(CC) $(OFILES_2D) -o $@

demo-energy: $(OFILES_ENERGY)
	$(CC) $(OFILES_ENERGY) -o $@ -lm

demo-constrained: $(OFILES_CONSTRAINED)
	$(CC) $(OFILES_CONSTRAINED) -o $@

clean:
	rm -f  $(ALL) $(OFILES_RANK_VERTICES) $(OFILES_1D) $(OFILES_2D) \
	                 $(OFILES_ENERGY) $(OFILES_CONSTRAINED)

# the output of "gcc -MM" goes here
...
```

Chapter 7

Allocating memory: `xmalloc()`

Prerequisites: *None*

7.1 ▪ Introduction

The C standard library's `malloc()` function, declared in the header file *stdlib.h*, allocates a contiguous block of memory of the requested size (in bytes) and returns a void pointer (cf. Section 4.3) that points to the beginning of the block. The request may fail if the requested amount is greater than what's available. In that case `malloc()` returns `NULL`.

The action to take when `malloc()` fails is dictated by the nature of the application. For instance, if a text editing program runs out of memory, the proper action would be to notify the user of the fact and give him the opportunity to save the work before exiting. Similarly, if the overnight execution of a lengthy scientific computation runs out of memory, it would be prudent that it saves its current state before quitting, so that it may be resumed later from where it was cut off rather than starting anew. At the other extreme, if the program has nothing worthwhile to save, then the proper action would be to issue an alert and then quit.

All of this book's programs fall in the latter category. Therefore we introduce a function `malloc_or_exit()` that serves as a front end to `malloc()`. It calls `malloc()` and checks the result. If `NULL`, then it prints a diagnostic to the `stderr` and calls `exit()` to terminate the program. Otherwise it returns the result to the caller. By implementing this wrapper around `malloc()`, we are freeing the user of the responsibility of checking explicitly for `malloc()`'s success—if the program is still alive after `malloc_or_exit()` returns, then `malloc()` must have succeeded. This simplifies and streamlines our programs substantially.

Furthermore, we introduce a function-like preprocessor macro, `xmalloc()`, which offers a simplified interface to `malloc_or_exit()`, as we shall see below. All memory allocation in the rest of this book is done through the `xmalloc()` macro.

7.2 ▪ A review of `malloc()`

Our *xmalloc* module relies on the standard library's `malloc()` function for allocating memory. A typical usage of `malloc()` is shown in Listing 7.1. There are quite a few ideas there, so let us go through it line by line (but not necessarily in sequential order).

Listing 7.1: A typical use of the standard library's `malloc()` function.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main(void)
4  {
5      int n = 10;
6      double *x = malloc(n * sizeof *x);
7      if (x == NULL) {
8          fprintf(stderr, "%s:line %d: malloc(%u) failed\n",
9              __FILE__, __LINE__, n * sizeof *x);
10         return EXIT_FAILURE;
11     }
12     for (int i = 0; i < n; i++)
13         x[i] = 1.0 / (i + 1);
14     for (int i = 0; i < n; i++)
15         printf("x[%d] = %g\n", i, x[i]);
16     free(x);
17     return EXIT_SUCCESS;
18 }
```

Line 6: A lot is happening on this line. The "**double** `*x`" part declares `x` as a pointer to **double**. Then "**sizeof** `*x`" determines the size of *what* `x` *points to*. Since `x` points to a **double**, then "**sizeof** `*x`" produces the size of the storage for a **double** (which is 8 bytes on my machine but may be different on yours). Then `malloc()` is called to request storage for `n` times the size of a **double**, where `n` is defined on the preceding line. Finally, the address of the memory block returned by `malloc()` is assigned to `x`.

Remark 7.1. A contiguous memory block capable of storing `n` doubles is just the right object to hold a (mathematical) vector of floating point numbers. This is the subject of Chapter 8.

Lines 7–11: If the call to `malloc()` fails—which can happen if there is insufficient memory available to meet the request—it returns `NULL`. We check for that on line 7, and if so, we print a message to `stderr` and return. Returning from `main()` ends the program. The value `EXIT_FAILURE`, which is defined in *stdlib.h*, is passed to the calling program (typically the user's terminal) to signal the program's failure.

A large program may be spread over several functions and several files, and may make numerous calls to `malloc()`. A helpful failure message will tell the user the point at which the failure occurred. In the case of a failure, our program prints a message like

```
malloc-demo.c:line 10: malloc(2115098112) failed
```

The message shows the file name and line number where `malloc()` failed, as well as the amount of memory requested, in bytes. The C preprocessor macros `__FILE__` and `__LINE__` embedded anywhere in a program take on the file name and line number of that location.[14]

[14]Since it may be difficult to discern in a typeset document, let me point out that each of the `__FILE__` and `__LINE__` symbols has *two leading and two trailing underscores*.

Line 12: We fill the allocated space with the sequence of numbers $\{\frac{1}{i+1}\}_{i=0}^{n-1}$. Note that the pointer x is being treated *exactly like a C array*. How is that?

The answer lies in the pointer arithmetic of Section 4.1. For one thing, x points to the beginning of the allocated memory block. For another thing, since x is a pointer to **double**, adding i to x will move forward by "i times the size of **double**" bytes in the memory space. Thus, "*(x + i)" will represent the value stored in the ith position. Accordingly, line 13 could have been written as "*(x + i) = 1.0 / (i + 1);". However, *(x + i) is the same as x[i], as we noted on page 22. This explains the form of line 13.

Remark 7.2. What we have learned here is that memory allocated by malloc() may be handled with a syntax identical to that of the built-in C arrays, although the two are *quite different objects* as far as C goes. What makes the identical syntax possible is the clever design of the pointer arithmetic.

Line 14: We go through the allocated memory and print out its contents. There is nothing new here, but observe again that the syntax to access the content of the ith location is just as if x were an ordinary C array.

Line 16: We call free(), which is declared in *stdlib.h* along with malloc(), to release the memory that was procured by malloc(). A well-designed program will have as many calls to free() as there are calls to malloc(). In fact, the allocated memory should be freed as soon as it is no longer needed so that the program may reclaim it for other use. There is no reason to hog a system resource if it's no longer needed.[15]

Modern operating systems automatically release all memory allocated by a program when the program terminates. In such systems, releasing memory explicitly just before the exit is redundant. Nevertheless, a fastidious attention to releasing all allocated memory before leaving the program is the mark of an experienced programmer, much in the same way that cleaning up the area before striking the tent is an indicator of a conscientious camper. There are tangible benefits as well, other than just making you feel good about yourself. Experience shows that attention to the details of allocation and freeing memory imposes a certain self-discipline which ultimately results in better structured and more elegant programs. Furthermore, if it happens that your program is imbedded later in a larger program, the place where your program used to exit is no longer a point of exit of the larger program. If your program does not clean up after itself, it can wreak havoc with the larger program.

As a case in point, think of your program as a solver of a static partial differential equation and the larger program as a time-stepper that simulates an evolving dynamics. At every one of the thousands of time-steps, the time-stepper calls your program to update its state. A memory leak in your program can bring the larger program to a halt. I know; I am speaking from experience.

Valgrind from <http://valgrind.org/> is an open source free software that can check your program for memory leaks. Invoked at the command-line as

```
$ valgrind  your-program's-name  your-program's-arguments
```

[15] This is especially important for programs that run over extended periods of time, such as web browsers, system daemons, and servers. Memory allocated and abandoned without freeing can grow over time and bring the system to standstill. Programs that suffer from this malady are said to "leak memory".

it will run the program under its control, and at the end will print a report indicating the number of times your program called `malloc()` and `free()` and whether there was a memory leak.

You will find an extensive discussion of memory management issues in Chapter 8 of *C Unleashed* [27]. You may be interested in the `memtrack` utility described there to track memory allocation and release in a complex program.

Line 17: The value of `EXIT_SUCCESS`, defined in *stdlib.h*, is passed to the calling program (typically the user's terminal shell) to signal the program's successful termination. Returning `EXIT_SUCCESS` from `main()` is equivalent to returning 0. The macro `EXIT_SUCCESS` exists for symmetry with the macro `EXIT_FAILURE`. There is no standard numerical code corresponding to the latter.

Line 1: The functions `printf()` and `fprintf()` are declared in the header file *stdio.h*.

Line 2: The functions `malloc()` and `free()`, as well as the macros `EXIT_SUCCESS` and `EXIT_FAILURE`, are declared in the header file *stdlib.h*.

Line 5: The value of `n`, defined here as 10, is used in the argument to `malloc()` and a couple of other places in the program. It is important to note that `n` need not have been a preassigned constant. With a little more effort, we could have arranged things so that the program would read the value of `n` from the command-line, for example. The point is, the vector `x` need not have a predetermined length. This is very much in contrast to C's native arrays, which can only have a priori fixed sizes. See Remark 4.6 on page 23 for details.

7.3 ▪ The program

This chapter's *xmalloc* module facilitates allocating and freeing memory, and is used for that purpose in almost every chapter in the rest of this book. The files *xmalloc.h* and *xmalloc.c* constitute the module's interface and implementation. The files *xmalloc-demo-1.c* and *xmalloc-demo-2.c* are two drivers for demonstrating some of the uses and features of the module. This project's directory contains

```
$ cd xmalloc
$ ls -F
xmalloc-demo-1.c  xmalloc-demo-2.c  xmalloc.c  xmalloc.h
```

We may automate the compilation with the help of a *Makefile* along the lines of the instructions in Chapter 6, but the current demos are so small that writing a *Makefile* for them may be an overkill. To compile them manually, we do

```
$ gcc -c -Wall -pedantic -std=c99 -O2 xmalloc.c
$ gcc -c -Wall -pedantic -std=c99 -O2 xmalloc-demo-1.c
$ gcc xmalloc.o xmalloc-demo-1.o -o xmalloc-demo-1
$ gcc -c -Wall -pedantic -std=c99 -O2 xmalloc-demo-2.c
$ gcc xmalloc.o xmalloc-demo-2.o -o xmalloc-demo-2
```

The only reason for specifying `-std=c99` is the use of the `fprintf()`'s `%zu` conversion specifier for printing values of type `size_t`. The `z` specifier was introduced in C99. See the discussion on page 6 for the C89 alternative.

Here is the transcript of an interactive session with the first demo:

```
$ ./xmalloc-demo-1
allocating 1000 bytes
memory freed
allocating 0 bytes
xmalloc-demo-1.c:line 18: malloc(0) failed
```

As we see, the program allocates 1000 bytes successfully and then frees it. Next, it attempts to allocate 0 bytes. Our program refuses such a request (what does it *mean* to allocate 0 bytes?), so it prints a diagnostic message and exits. (See Part 7.1 of Section 7.5 for more on this.)

Here is the transcript of an interactive session with the second demo:

```
./xmalloc-demo-2
     1: 1000000000 bytes allocated
     2: 1000000000 bytes allocated
▶ (the next 140730 lines deleted)
140734: 1000000000 bytes allocated
140735: 1000000000 bytes allocated
xmalloc-demo-2.c:line 10: malloc(1000000000) failed
```

This demo allocates memory chunks of 1,000,000,000 bytes (1 gigabyte) in an unending **`while`**-loop without freeing them. It prints a line upon each pass through the loop. After stepping through the loop 140,735 times, my computer's memory is exhausted; therefore the program exits with a diagnostic.

Does my computer truly have 140,735 gigabytes of memory? Not a chance! What we see here is a Linux artifact—I am running this program on a computer with a 64-bit Linux operating system—which allocates memory generously in a *virtual memory space* far beyond what is physically available, in the hope that not all of the requested memory will actually be needed. Under a more conservative operating system the program may have stopped in fewer than 8 iterations since I have only 8 gigabytes of physical memory on this computer.

7.4 ▪ The interface and the implementation

The *xmalloc* module is needed in almost all of this book's projects; therefore I expect that every reader will go through this chapter early in their studies. For that reason, I will devote more than the usual amount of attention to details here. In particular, in the following two subsections I will present essentially the entire contents of the files *xmalloc.c* and *xmalloc.h*. You will make a slight adjustment to my *xmalloc.c* in the *Projects* section.

7.4.1 ▪ *xmalloc.c*: The implementation

Listing 7.2 gives a preliminary version of the file *xmalloc.c* which implements our *xmalloc* module. It defines a function `malloc_or_exit()` which receives, as its first argument, the number of bytes of memory to be allocated, and as the second and third arguments the file name and line number whence it was invoked. If the allocation is successful, it returns a pointer to the start of the allocated memory; otherwise it prints a message of the sort

```
foo.c:line 123: malloc() of 10000 bytes failed
```

Listing 7.2: This is a preliminary version of the file *xmalloc.c*. The function `malloc_or_exit()` attempts to allocate `nbytes` of memory. If successful, it returns a pointer to that memory; otherwise it prints a message and exits the program. See Part 7.1 of Section 7.5 regarding a subtle flaw and how to fix it.

```
1  #include <stdio.h>
2  #include "xmalloc.h"
3  void *malloc_or_exit(size_t nbytes, const char *file, int line)
4  {
5      void *x;
6      if ((x = malloc(nbytes)) == NULL) {
7          fprintf(stderr, "%s:line %d: malloc() of %zu bytes failed\n",
8                  file, line, nbytes);
9          exit(EXIT_FAILURE);
10     } else
11         return x;
12 }
```

to the `stderr` and then calls `exit()` to quit the program. Note the two occurrences of the void pointer in that listing, and be sure to understand their purpose. Refer to Section 4.3 for an explanation of void pointers.

Here is a typical call to `malloc_or_exit()`:

```
double *x;
x = malloc_or_exit(100 * sizeof *x, __FILE__, __LINE__);
... use x ...
free(x);
```

This allocates a contiguous area of memory capable of holding 100 numbers of the type **double**. The C preprocessor macros `__FILE__` and `__LINE__` take on the file name and the line number of where they are invoked, as noted on page 46.

The call to `malloc_or_exit()` may be simplified by noting that although the value passed to its first argument will vary in general from one instance to the next, the second and third arguments are always `__FILE__` and `__LINE__`. This motivates the introduction of the preprocessor macro[16]

```
#define xmalloc(nbytes) malloc_or_exit((nbytes), __FILE__, __LINE__)
```

which expands the expression `xmalloc(nbytes)` into a full-fledged call to the function `malloc_or_exit()`. The caller's interface will look like this:

```
double *x = xmalloc(100 * sizeof *x);
... use x ...
free(x);
```

and that's as simple as it's going to get.

From now on we will call the `xmalloc()` wrapper instead of `malloc_or_exit()` (or `malloc()`) to allocate memory in our programs.

Remark 7.3. The memory fetched by `malloc()` can be resized using the standard library's `realloc()` function. If a call to `realloc()` fails, it returns `NULL` just like

[16]Here I am violating the C tradition of all-capital names for macros. The all-caps `XMALLOC`, however, looks too ungainly to me; therefore I will break from the tradition in favor of the gentler lowercase `xmalloc`.

Listing 7.3: The header file *xmalloc.h* provides the *xmalloc* module's interface.

```
1 #ifndef H_XMALLOC_H
2 #define H_XMALLOC_H
3 #include <stdlib.h>
4 void *malloc_or_exit(size_t nbytes, const char *file, int line);
5 #define xmalloc(nbytes)  malloc_or_exit((nbytes), __FILE__, __LINE__)
6 #endif  /* H_XMALLOC_H */
```

`malloc()`; therefore for completeness it would be useful to have an `xrealloc()` to accompany `xmalloc()`. However, we don't have a use for `realloc()` in this book; therefore I will not explore that idea any further.

7.4.2 ▪ *xmalloc.h*: The interface

The header file *xmalloc.h*, shown in its entirety in Listing 7.3, provides the interface to our *xmalloc* module. A few comments on its structure are in order:

Lines 1, 2, and 6: These are the header file's so-called *header guards* or *#include guards*. Their purpose is to prevent unintended multiple inclusions of this header file in other units. To illustrate a case of multiple inclusion in the absence of *#include guards*, consider two header files *header1.h* and *header2.h* and a program file *prog.c* that **`#include`**s both of them. If *header2.h* also **`#include`**s *header1.h*, then *header1.h* will be read twice into *prog.c*, leading to potential complications.

If a header file is equipped with *#include guards*, as is our *xmalloc.h*, then this is what happens. When *xmalloc.c* **`#include`**s *xmalloc.h* for the first time, line 1 determines that the symbol `H_XMALLOC_H` is undefined; therefore control passes to line 2, where `H_XMALLOC_H` gets defined. The rest of *xmalloc.h* is read as usual.

Should *xmalloc.c* attempt to **`#include`** *xmalloc.h* for a second time, line 1 will determine that the symbol `H_XMALLOC_H` is already defined; therefore control will pass to line 6, thus bypassing the main body of *xmalloc.h*.

Every header file should be equipped with #include guards. The guard symbol, which is `H_XMALLOC_H` in our case, is arbitrary, but it should be unique within all header files of the entire project. The usual way of creating such a unique symbol is by basing it on the name of the corresponding header file. Since our header file is named *xmalloc.h*, any of the choices `XMALLOC`, `XMALLOC_H`, `H_XMALLOC_H`, or variations thereof, will serve the purpose. The second one is the traditional choice in C. I prefer the third one since the leading `H` avoids an (admittedly remote) clash with IEEE's POSIX standard, which reserves symbol names beginning with `E` for its own use.

The comment on line 6 is not required; its sole purpose is to remind one that the **`#endif`** on that line corresponds to the **`#ifndef`** `H_XMALLOC_H` on line 1. In a short header file such as our *xmalloc.h*, this is rather superfluous, but in a header file that spans hundreds of lines, the reminder is a nice touch.

Remark 7.4. It is a common beginner's mistake to name the *#include guard* symbols with a leading underscore, such as `_XMALLOC`, mimicking those in the C standard library. The C standard, however, reserves preprocessor symbols that begin

with an underscore *for its own private use*. If you name *your* symbols with a leading underscore, you run the risk of stepping over your compiler's internals. That's very bad. Don't do it!

Line 3: At first sight it may look odd why we have **#include** <stdlib.h> in *xmalloc.h* while **#include** <stdio.h> in *xmalloc.c*. The reason for including *stdio.h* in *xmalloc.c* is to provide a prototype for `fprintf()` that occurs in that file. There is no `fprintf()` in *xmalloc.h* and hence no need for *stdio.h* there. On the other hand, we include *stdlib.h* in *xmalloc.h* because it defines the symbol `size_t` which is needed in *xmalloc.h*.[17] There is yet another—and not so obvious—minor benefit of including *stdlib.h* in *xmalloc.h*. Programs that call `xmalloc()` to allocate memory eventually will call `free()` to release that memory. When they include *xmalloc.h*, they receive *stdlib.h* (which declares the prototype of `free()`) as a bonus; therefore they need not include it separately.

7.5 ▪ *Project Xmalloc*

Part 7.1. The function `malloc_or_exit()` in Listing 7.2 allocates `nbytes` bytes of memory and returns that memory's address. But what should it return if `nbytes` is zero? The C99 standard has the following to say:

> **Section 7.20.3:** If the size of the space requested is zero, the behavior is implementation-defined: either a null pointer is returned, or the behavior is as if the size were some nonzero value, except that the returned pointer shall not be used to access an object.

In short, allocating zero bytes is ill-defined, so we might as well treat it just as when `malloc()` fails. Modify `malloc_or_exit()` to account for this.

Part 7.2. Implement the adjustment suggested in the previous part. Then write a program, let's say *xmalloc-demo-1.c*, that does the following:

1. calls `xmalloc()` to allocate 1000 bytes of memory and then prints a message saying that the memory was allocated;
2. frees that memory and then prints a message saying that the memory was freed;
3. calls `xmalloc()` to allocate 0 bytes of memory and then prints a message saying that the memory was allocated; and
4. frees that memory and then prints a message saying that the memory was freed.

Will the program print 4 lines? See the transcript of the interactive session on page 49.

Part 7.3. Write a program, let's say *xmalloc-demo-2.c*, that allocates memory in chunks of 1,000,000,000 bytes (1 gigabyte) and abandons them (that is, does not free them), in a nonterminating (infinite) loop. Print a message on each pass. Will the program run forever? See the transcript of the interactive session on page 49.

[17] The symbol `size_t` is also defined in the standard header file *stddef.h*; therefore one could include it instead of *stdlib.h* if the only objective were to define `size_t`.

Chapter 8

Dynamic memory allocation for vectors and matrices: *array.h*

Prerequisite: *Chapter 7*

8.1 ▪ Introduction

In this chapter we introduce a preprocessor trick to construct vectors, matrices—and even higher-dimensional array-like objects—in a type-generic way in standard C.[18] The preprocessor macro `make_matrix()`, for instance, constructs a matrix of any type. Thus, to construct 5×7 matrices of the types **`int`**, **`double`**, and "**`struct`** `point`", we do

```
int **a;
double **x;
struct point {double x; double y;} **p;
int m = 5, n = 7;
make_matrix(a, m, n);
make_matrix(x, m, n);
make_matrix(p, m, n);
x[0][0] = 3.14159;    // use x, etc.
free_matrix(a);
free_matrix(x);
free_matrix(p);
```

Observe how the three matrices of varying types are constructed and freed through an identical interface. There is also a `make_vector()` to construct type-generic vectors, and `make_3array()` and `make_4array()` to construct type-generic three- and four-dimensional arrays. All these are bundled in a header file named *array.h*. There is no associated *array.c* file.

Preprocessor macros are not the most elegant part of the C programming language, and they are best avoided if the job can be done without them. Unfortunately, type-generic construction of arrays, as laid out above, is impossible in C89 and C99 without the help of preprocessor macros since C lacks polymorphism. On the balance, the absolutely simple user-interfaces and the versatility of these macros far outweigh the somewhat ungainly code that resides in *array.h*. At any rate, what we have here is better than any alternatives of which I know. We rely on the macros provided in *array.h* throughout the rest of this book to create and destroy vectors and matrices. Thankfully, we won't have to look inside *array.h* every time we use it.

[18]The trick works equally well in C89 and C99.

8.2 ▪ Constructing vectors of arbitrary types

The `xmalloc()` wrapper of Chapter 7 provides a convenient way for creating vectors of various lengths and types. With simple wrappers such as[19]

```
int *make_ivector(size_t n)          // provisional
{
    int *v = xmalloc(n * sizeof *v);
    return v;
}
double *make_dvector(size_t n)       // provisional
{
    double *v = xmalloc(n * sizeof *v);
    return v;
}
```

we may call `make_ivector(n)` and `make_dvector(n)` to make vectors of length `n` of the types **`int`** and **`double`**.

If this were the end of the story, the functions above could justly be criticized on grounds that they make things more complicated than necessary. After all, allocating memory for a vector of `n` double-precision floating point numbers can be as simple as

```
double *v = malloc(n * sizeof *v);
```

where `make_dvector()` and `xmalloc()` don't come into the play at all. However, this is *not* the end of the story. What we have so far is an infrastructure upon which we are going to build extensively. The significance of this will become apparent when we construct type-generic vectors, matrices, and higher-dimensional arrays later in this chapter.

Before doing that, however, let us revisit the two functions defined above. There is something unsatisfactory in their approach since, continuing along those lines, we will need to write a special-purpose vector allocation function for *every new data type* that comes along. That's annoying, particularly since the number of potential data types is infinite. It would be more elegant having a single function capable of constructing vectors of *all types*. Although such a function is infeasible—there is no polymorphism in C—the equivalent effect is achieved easily through a function-like preprocessor macro

```
#define make_vector(v,n)   ((v) = xmalloc((n) * sizeof *(v)))
```

which is type-agnostic and may be used to allocate memory for vectors of *any* type.[20] In the following code fragment, we apply the `make_vector()` macro to make a vector of length 100 of type **`double`**:

```
double *x;
make_vector(x, 100);
for (int i = 0; i < 100; i++)
    x[i] = 1.0 / (1 + i);
free_vector(x);
```

In the next example, we apply *the same* `make_vector()` macro to make a vector of length 30 of type "**`struct`** `point`":

[19] I have marked these functions "provisional" since the `make_vector()` macro, to be introduced shortly, does the same job and much more!

[20] The plethora of parentheses in this macro definition is standard fare. Consult the chapter on preprocessor macros in your C reference book for an explanation. Also see footnote 16 on page 50 regarding a break with tradition.

```
struct point {double x; double y;} *p;
make_vector(p, 30);                        // make a vector of 30 points
p[0]→x = 3.14;  p[0]→y = 2.78;             // define p[0]'s coordinates
p[1]→x = 12.1;  p[1]→y = -0.3;             // define p[1]'s coordinates
... etc. ...
free_vector(p);
```

What is `free_vector()`? It frees the memory allocated by `make_vector()`. In principle, freeing that memory is a matter of calling the standard library's `free()`, as in "`free(x)`" or "`free(p)`" in the two examples above. Nevertheless, we introduce the `free_vector()` macro as a simple wrapper around `free(p)`, for the sake of symmetry with `make_vector()`. And as long as we are at it, we go one step further and arrange things so that after freeing the memory, the corresponding pointer is set to `NULL`. This is an unnecessary and insignificant frill, but there are those who are passionate about `NULL`-ing freed pointers. This should please them:

```
#define free_vector(v)  do { free(v); v = NULL; } while (0)
```

The `free(v)` and `v = NULL` parts require no explanation, but the **do** and **while**`(0)` would definitely look strange if you are seeing this well-known construction for the first time. Let me explain.

Preprocessor macros consisting of multiple statements must be handled with care. Consider a hypothetical macro consisting of two statements:

```
#define bad_macro  statement1; statement2
```

Then observe that the code fragment

```
if (test1)
    bad_macro;
else
    something_else;
```

expands to

```
if (test1)
    statement 1;
    statement 2;
else
    something_else;
```

which is syntactically incorrect—braces are missing. Adding braces to the macro

```
#define bad_macro    { statement1; statement2; }
```

does not help because now the code fragment expands to

```
if (test1) {
    statement 1;
    statement 2;
}; else
    something_else;
```

which is still incorrect—the semicolon before the **else** shouldn't be there. However, setting

```
#define good_macro  do { statement1; statement2; } while (0)
```

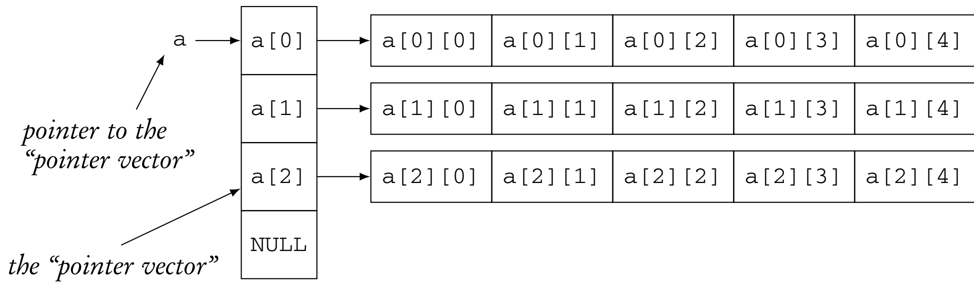


Figure 8.1: The schematic representation of the memory layout for a 3×5 matrix. The "pointer vector" that appears as a vertical column on the left is a vector of pointers to the matrix's rows.

does the trick. The `while(0)` condition ensures that the block inside the braces is executed exactly once. The code fragment expands to

```
if (test1)
    do {
        statement 1;
        statement 2;
    } while (0);
else
    something_else;
```

which is just right.

8.3 ▪ A scheme for dynamically allocated matrices

Figure 8.1 is a schematic representation of memory layout for a dynamically allocated 3×5 matrix. The matrix rows are viewed as vectors of length 5 obtained by calls to `make_vector()`. The addresses of the first elements of those vectors are stored in a "column" vector of length 4 shown along the diagram's left edge. We call that the *pointer vector* because its elements are pointers to the row vectors. The last element of the pointer vector is set to `NULL`; it serves as a *sentinel* that marks the end of the vector. Compare Figure 8.1 with Figure 4.1 on page 26 concerning C's storage scheme for the command-line arguments. They reflect identical ideas.

The identifier `a`, which serves as the name of the matrix, holds the address of the pointer vector's first element. Since the elements of the pointer vector are pointers, then `a` is a *pointer to pointer*.

The elements of the pointer vector are `a[i]`, `i` $= 0, \dots, 3$; therefore the address of the ith row is `a[i]`, and hence the values stored in that row are `a[i][j]`. We see that such dynamically allocated matrices and C's native two-dimensional arrays have identical syntax for addressing the matrix's elements. However, their internal storage and representations can be entirely different. *Matrices constructed here should not be confused with C's native two-dimensional arrays.*

The provisional[21] `make_dmatrix()` function defined below implements the scheme described above for constructing an $m \times n$ matrix of type **`double`**:

[21] I have marked this function "provisional" since the `make_matrix()` macro developed in the next section does the same job and much more!

```
double **make_dmatrix(size_t m, size_t n)    // provisional
{
    double **a;
    make_vector(a, m + 1);                   // make the pointer vector
    for (size_t i = 0; i < m; i++)
        make_vector(a[i], n);                // make the row vectors
    a[m] = NULL;
    return a;
}
```

Thus, to create a 10×10 Hilbert matrix, we do[22]

```
double **H;
int n = 10;
H = make_dmatrix(n, n);
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        H[i][j] = 1.0 / (1 + i + j);
... work with H ...
free_dmatrix(H);
```

The `free_dmatrix(H)`, which frees the memory allocated by `make_dmatrix()`, looks suspicious at first—how does `free_dmatrix()` know how many rows there are to free? As it turns out, `free_dmatrix()` need not know the number of rows. What it will do is walk down the pointer vector and free the memory pointed to by each of its elements. When it reaches the pointer vector's `NULL` sentinel, it will know that all the rows have been freed. (That's the sole reason for building a sentinel into the pointer vector.) All there remains is to free the pointer vector and it's done. Here is the implementation:

```
void free_dmatrix(double **a)        // provisional
{
    if (a ≠ NULL) {
        for (size_t i = 0; a[i] ≠ NULL; i++)
            free_vector(a[i]);
        free(a);
    }
}
```

Remark 8.1. Passing `NULL` to `free_dmatrix()` is safe, as the function checks for it. In that respect, `free_dmatrix()` behaves like the standard library's `free()` function which is also a no-op against `NULL`.

Moreover, note that without that check, passing `NULL` to `free_dmatrix()` will have crashed it because inside the **`for`**-loop it will have attempted to access the nonexistent `a[i]`.

8.4 ▪ Constructing matrices of arbitrary types

The previous section's `make_dmatrix()` and `free_dmatrix()` functions suffer from the same malady as `make_dvector()` did in the section prior to that—they are

[22]The Hilbert matrix is a square matrix with entries $h_{ij} = 1/(1+i+j)$, where the indices i and j are counted from zero.

limited to constructing and freeing matrices of the type **double**. To make a matrix of the type **int**, for instance, we will have to write another version.

In Section 8.2 we overcame the problem in the case of vectors with the help of a preprocessor macro. In this section we extend that preprocessor trick to matrices. This is quite straightforward other than the issue of handling the loop counter `i` in the function `make_dmatrix()` defined on page 57. The scope of that counter is limited to the body of the function, and as such it is invisible to the outside world. Converting the function to a preprocessor macro, however, exposes its innards and poses a potential for a clash between its loop counter and another identifier of the same name which may be lurking in the code. The way around this is to use a *reserved identifier name* instead of a generic `i` and alert the macro's users[23] of its presence. I call the counter "`make_matrix_loop_counter`" since the close association with the name "`make_matrix`" makes it unlikely that a user will encroach upon it inadvertently. With this out of the way, here is the promised macro:

```
#define make_matrix(a, m, n) do {                                  \
    size_t make_matrix_loop_counter;                               \
    make_vector(a, (m) + 1);                                       \
    for (make_matrix_loop_counter = 0;                             \
            make_matrix_loop_counter < (m);                        \
            make_matrix_loop_counter++)                            \
        make_vector((a)[make_matrix_loop_counter], (n));           \
    (a)[m] = NULL;                                                 \
} while (0)
```

In the same way, we convert the function `free_dmatrix()` to a macro:

```
#define free_matrix(a) do {                                        \
    if (a ≠ NULL) {                                                \
        size_t make_matrix_loop_counter;                           \
        for (make_matrix_loop_counter = 0;                         \
                (a)[make_matrix_loop_counter] ≠ NULL;              \
                make_matrix_loop_counter++)                        \
            free_vector((a)[make_matrix_loop_counter]);            \
        free_vector(a);                                            \
        a = NULL;                                                  \
    }                                                              \
} while (0)
```

The backslashes ("\") indicate line continuation in a preprocessor macro. I have aligned them vertically purely for aesthetic reasons; they don't have to be aligned. I have used the reserved identifier `make_matrix_loop_counter` in both macros; there is no reason to introduce two different reserved words. Additionally, I have added `a = NULL` at the end of `free_matrix()` for the same reason we did so in the case of `free_vector()`.

On the surface, these macros are ungainly. Nevertheless they are quite general and offer particularly clean interfaces, and ultimately that's what matters. Here is the previous section's Hilbert matrix, constructed with the help of `make_matrix()`:

```
double **H;
int n = 10;
make_matrix(H, n, n);    // was H = make_dmatrix(n, n);
```

[23] The "users" includes you!

Listing 8.1: An outline of the file *array.h*.

```
1 #ifndef H_ARRAY_H
2 #define H_ARRAY_H
3 #include "xmalloc.h"
4  ▶ #define make_vector(...
5  ▶ #define free_vector(...
6  ▶ #define make_matrix(...
7  ▶ #define free_matrix(...
8 #endif /* H_ARRAY_H */
```

```
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        H[i][j] = 1.0 / (1 + i + j);
... work with H ...
free_matrix(H);
```

Only one line (as marked) has changed relative to the previous version.

8.5 ▪ *Project array.h*

Create a directory, say *vector-and-matrix*, to hold this chapter's material. Our macros rely on Chapter 7's *xmalloc* module for allocating memory. If you organize your files as suggested in Chapter 2, your directory will look like this:

```
$ cd vector-and-matrix
$ ls -F
array.h  hilbert-matrix.c  xmalloc.c@  xmalloc.h@
```

where the "`@`" decoration at the end of a file name indicates a symbolic link, as explained in Chapter 2. The details of the files *array.h* and *hilbert-matrix.c* are described below.

Part 8.1. Put the definitions of the macros `make_vector()`, `free_vector()`, `make_matrix()`, and `free_matrix()` in a file *array.h*, and add *#include guards*, as shown in an outline form in Listing 8.1. See page 51 regarding the significance of *#include guards*.

Part 8.2. Listing 8.2 provides an outline of a file named *hilbert-matrix.c*. In it we have a function named `hilbert_matrix()` which constructs and returns the address of an $n \times n$ Hilbert matrix for a given n. (See footnote 22 on page 57 for the definition of a Hilbert matrix.) The function `hilbert_matrix()` is called from within `main()` to build an 8×8 Hilbert matrix. The program exits after printing the matrix and freeing the allocated memory.

Supply the details, compile, and execute the program. Its should print something like this:

```
1.000   0.500   0.333   0.250   0.200   0.167   0.143   0.125
0.500   0.333   0.250   0.200   0.167   0.143   0.125   0.111
0.333   0.250   0.200   0.167   0.143   0.125   0.111   0.100
0.250   0.200   0.167   0.143   0.125   0.111   0.100   0.091
0.200   0.167   0.143   0.125   0.111   0.100   0.091   0.083
0.167   0.143   0.125   0.111   0.100   0.091   0.083   0.077
0.143   0.125   0.111   0.100   0.091   0.083   0.077   0.071
0.125   0.111   0.100   0.091   0.083   0.077   0.071   0.067
```

Listing 8.2: An outline of *hilbert-matrix.c*.

```
▶ the necessary #includes
double **hilbert_matrix(int n)
{
    ▶ call make_matrix() to make and return the
      address of an nxn Hilbert matrix
}
int main(void)
{
    double **H;
    int n = 8;
    H = hilbert_matrix(n);
    ▶ print the matrix
    ▶ free the matrix
    return 0;
}
```

Listing 8.3: A type-generic implementation of the preprocessor macro `print_vector()`. We use the symbol `print_vector_loop_counter` for a loop counter. The users of this macro should be told that this is a reserved word and should not be interfered with.

```
#define print_vector(fmt, v, n) do {                          \
    size_t print_vector_loop_counter;                          \
    for (print_vector_loop_counter = 0;                        \
                print_vector_loop_counter < (n);               \
                print_vector_loop_counter++)                   \
        printf(fmt, (v)[print_vector_loop_counter]);           \
    putchar('\n');                                             \
} while (0)
```

Listing 8.4: Code fragment showing a sample usage of the `print_vector()` macro. This should print "`   1.000    0.500    0.333    0.250    0.200`".

```
    double *v;
    int i, n = 5;
    make_vector(v, n);
    for (i = 0; i < n; i++)
        v[i] = 1.0 / (1 + i);
    print_vector("%7.3f ", v, n);
    free_vector(v);
```

Part 8.3. [optional] Add the type-generic `print_vector()` and `print_matrix()` macros in *array.h* for printing vectors and matrices to `stdout`. I have shown my implementation of `print_vector()` in Listing 8.3 and a sample usage in Listing 8.4.

I will leave it to you to write `print_matrix()`. You may call `print_vector()` from within `print_matrix()` to print each row.

Part 8.4. [optional] Revise your program of Part 8.2 to use `print_matrix()` to print the Hilbert matrix.

Part 8.5. [optional] Extend *array.h* by macros for making and freeing three- and four-dimensional arrays. Then you should be able to do

```
double ***C;
double ****D;
make_3array(C, 3, 5, 10);
make_4array(D, 3, 3, 3, 3);
C[0][0][0] = 1;
D[0][0][0][0] = 1;
... etc ...
free_3array(C);
free_4array(D);
```

Hint: Think of a three-dimensional array as a “column” vector whose entries are the addresses of matrices. Similarly, think of a four-dimensional array as a “column” vector whose entries are the addresses of three-dimensional arrays.

Chapter 9

Reading lines: `fetch_line()`

Prerequisites: None

9.1 ▪ Introduction

Some of this book's projects involve moderately complex configuration parameters and data. For instance, the truss solver of Chapter 19 requires information about the truss's geometry, loads, supports, and materials. The evolution simulator of Chapter 17 needs information about the size of the "world", the initial state of the animals, their genetic structures and energy content, among other things.

How does one convey such information to a C program? Hard-coding the parameters and data into a program would be rather primitive because any change—for instance, an adjustment of a truss's loads—will require editing and recompiling the program. There is much greater flexibility and freedom if the data stands separate from the program, typically in a text file supplied by the user. The program will read the data from the file and perform the required tasks. Changing the data will not require recompiling the program.

In this chapter we implement a `fetch_line()` function whose purpose is to retrieve *the next available nonempty line* from an input stream. Specifically, after reading a line, it trims it of comments and leading and trailing whitespace. If what remains is a string of zero length, it goes on to read the next line and repeat the process; otherwise it returns a pointer to the first character of the trimmed string. Upon the end of the input or an error condition, it returns `NULL`.

9.2 ▪ Reading one line at a time with `fgets()`

Underlying our `fetch_line()` is the standard library's `fgets()` function which reads one line of text from an input stream and stores it in an array of **`char`**—commonly referred to as a *buffer*—that the user provides. In the following minimal illustration, we call `fgets()` to read one line from the `stdin`:

```
char buf[12];
fgets(buf, 12, stdin);
```

The buffer's length (12 in this case) is passed to `fgets()` to let it know how much room it has to work with.[24] It reads *at most* 11 characters, one character at a time, from the

[24] A buffer length of 12 is much too small for most applications—100 or 128 would be more realistic—but 12 is a good size for this section's drawings/illustrations.

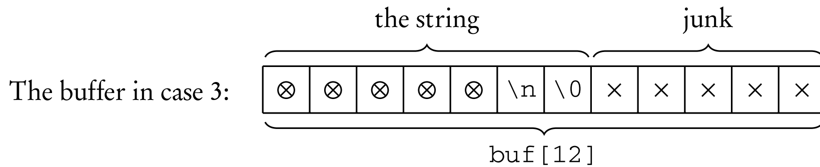
stream and stores them in the buffer. Reading stops when any of the following occurs:

1. The input stream ends or an error occurs.
2. It reaches the limit of 11 characters.
3. It encounters the newline (`\n`) character. It stores the newline in the buffer.

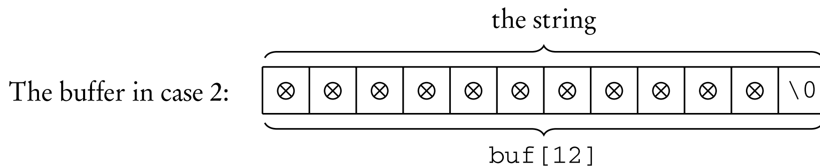
In any case, it inserts an ASCII `NUL` (`\0`) character after the last character read[25] to make the contents of `buf` into a properly null-terminated string.

In cases 2 and 3, `fgets()` returns a pointer to `buf`, which is rather redundant, because we already know where `buf` lives. In case 1 it returns the `NULL` pointer[26], which is significant—it tells us that there is nothing more to read.

At the end of case 3, the buffer looks like this:



where a ⊗ indicates any character other than `\0` and `\n`, and a × indicates any character at all. At the end of case 2 the buffer looks like this:



where the character before last may or may not be the newline character.

For the benefit of those readers who have not had much experience with the `fgets()` function, I have provided a complete and self-contained demo program, called *fgets-demo.c*, in Listing 9.1 to illustrate its use. The program reads one line at a time from the `stdin` and writes what it has read to the `stdout` until the end of input is reached. Examine it and be sure to understand every detail. Then compile it

```
$ gcc -Wall -pedantic -std=c89 -O2 fgets-demo.c -o fgets-demo
```

to produce the executable *fgets-demo*. To test, pass the contents of any text file to *fgets-demo* to see what it does. A good candidate for a text file is the program *fgets-demo.c* itself:

```
$ ./fgets-demo <fgets-demo.c
```

If all is well, the program should print the exact contents of *fgets-demo.c* to the screen.

Remark 9.1. In *fgets-demo.c* the buffer's length is obtained from the value of the preprocessor macro `BUFLEN` which I have arbitrarily set to 16. Defining the buffer length as a preprocessor macro makes it possible to change the length without delving too deeply into the innards of the program. Note that some of *fgets-demo.c*'s lines are significantly longer than 16; therefore `fgets()` will read those lines piecemeal. Nevertheless, the program's output will be identical to the input. Study the three cases described at the beginning of this section, and see if you can explain why that is so.

[25]That's the reason for the maximum of 11 rather than 12 characters.

[26]The ASCII `NUL` character has nothing to do with C's `NULL` pointer. Be sure not to confuse the two.

Listing 9.1: The file *fgets-demo.c* is a self-contained program that demonstrates the use of the standard library's `fgets()` function. The program reads from the `stdin` and writes to the `stdout`.

```
1 #include <stdio.h>
2 #define BUFLEN 16
3 int main(void)
4 {
5     char buf[BUFLEN];
6     while (fgets(buf, BUFLEN, stdin) ≠ NULL)
7         printf("%s", buf);
8     return 0;
9 }
```

9.3 ▪ Trimming whitespace and comments

Data files are written by people, and people like having a certain amount of leeway in how they format their writings. Among the most basic desiderata are the ability to

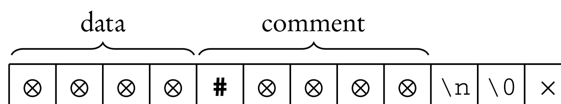
- indent lines;
- leave any number of spaces between data items within a line;
- insert any number of blank lines to emphasize data groupings; and
- insert notes and comments which should be ignored by the program.

To implement such a wish list, we need to make precise the meanings of the terms “space”, “blank line”, and “comment”.

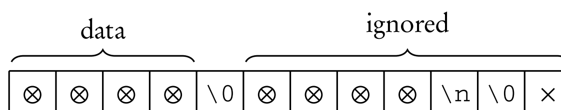
9.3.1 ▪ Comments

It's quite common, and convenient, to include comments in a data file in the way of documentation and clarification. The program that reads the data file should skip over the comments. It's up to the program's designer—that's you—on how comments are marked. In a C program, for instance, comments are enclosed between the `/*` and `*/` delimiters. Such comments may span multiple lines. C++ and C99 allow one-line comments marked by the `//` marker. The compiler ignores everything from that mark to the end of line. The `%` character in TeX and LaTeX, the `;` character in Lisp, and the **#** character in Unix shell scripts mark one-line comments in the same way. I will use the **#** character for commenting in our data files.

Ignoring everything after a **#** character to the end of line is quite simple. Suppose that the contents of a buffer fetched by `fgets()` looks like this:



To truncate the line at the **#** mark, simply overwrite the **#** by `\0`:



Since the ASCII NUL character (\0) marks the end of a string, anything that comes after it is immaterial.

9.3.2 ▪ Whitespace

When typing text into a text editor, you press the keyboard's "space-bar" key to enter spaces between words; you may press the keyboard's "Tab" key to indent the lines of your C program; you press the "Enter" key to insert a new line. These keys generate what are known as *whitespace characters*. Words in a text document are typically separated by whitespace.

There are other whitespace characters, although the common typewriter and computer keyboards don't have dedicated keys for them. The C standard recognizes at least six whitespace characters (although it doesn't call them "whitespace"): an ordinary space ( ), form-feed (\f), newline (\n), carriage return (\r), horizontal tab (\t), and vertical tab (\v).[27] It is important for our purposes to note that the ASCII NUL character (\0) is *not* a whitespace character.

The C standard library provides the isspace() function which receives an object as its argument and returns *true* if it is deemed whitespace, and *false* otherwise. The function isspace() and a dozen or so other character-testing functions are declared in the header file *ctype.h*; you need to **#include** it if you are going to use them. All those functions expect an **unsigned char** (or EOF) for their arguments; therefore the proper way for checking for a whitespace character requires a cast, as in

```
if (isspace((unsigned char)c))
    ... do something ...
```

9.4 ▪ The program

As promised earlier, our fetch_line() reads the next available line from an input stream, trims it of comments and leading and trailing whitespace, and returns a pointer to the trimmed string if the string is nonempty; otherwise it goes on to the next line and repeats the process.

The file *fetch-line.c* contains the implementation of the function fetch_line(). The header file *fetch-line.h* is the interface. The pair of files *fetch-line.[ch]* constitute our *fetch-line* module. A third file, *fetch-line-demo.c*, contains a *driver* to demo that module. Thus, this project's directory will contain

```
$ cd fetch-line
$ ls -F
fetch-line-demo.c  fetch-line.c  fetch-line.h  fgets-demo.c
```

The stand-alone test program *fgets-demo.c* was described in Section 9.2. The other three program files are described below. We may automate the program's compilation through a *Makefile* along the lines of the instructions in Chapter 6, but the current program is so small that writing a *Makefile* for it may be an overkill. To compile it manually, we do

```
$ gcc -c -Wall -pedantic -std=c89 -O2 fetch-line.c
$ gcc -c -Wall -pedantic -std=c89 -O2 fetch-line-demo.c
$ gcc fetch-line.o fetch-line-demo.o -o fetch-line-demo
```

[27] Additional characters may pass for whitespace depending on your *locale* environment. You will have to consult the C standard if you want more details on this.

Listing 9.2: The header file *fetch-line.h* is *fetch-line* module's interface.

```
1 #ifndef H_FETCH_LINE_H
2 #define H_FETCH_LINE_H
3 #include <stdio.h>
4 char *fetch_line(char *buf, int buflen, FILE *stream, int *lineno);
5 #endif /* H_FETCH_LINE_H */
```

You may change the `std=c89` to `std=c99` if you wish. The program is compatible with both C89 and C99.

Here is a transcript of an interactive session with the program:

```
$ ./fetch-line-demo <fgets-demo.c
trimmed line   3: int main(void)
trimmed line   4: {
trimmed line   5: char buf[BUFLEN];
*** reading error: input line 6 too long for fetch_line's buf[40]
```

As you see, we are feeding the file *fgets-demo.c* (which we encountered in Listing 9.1 on page 65) to our demo program.[28] Compare the program's output shown here to the contents of *fgets-demo.c*. From `fetch_line()`'s point of view, the first two lines of the input are comments(!) since each begins with the **#** character; therefore they are discarded. The third line receives `fetch_line()`'s approval; hence it duly prints it, prefacing it with the phrase "`trimmed line 3:`". Line 4, consisting of a single `{` character, is also printed without change. Line 5's leading whitespace is trimmed away, and what remains is printed. Line 6 is not acceptable to `fetch_line()`; it is too long to fit in a buffer of length 40 which it is given to work with. Therefore it prints an error message and quits. If we increase the buffer size sufficiently (see the `BUFLEN` preprocessor macro in Part 9.1 of Section 9.6), the program will read the file all the way to the end.

9.5 ▪ The files *fetch-line.[ch]*

The files *fetch-line.h* and *fetch-line.c* contain our *fetch-line* module's interface and implementation, respectively. The contents of *fetch-line.h* are shown in their entirety in Listing 9.2. That file's sole purpose is to declare the prototype of the function `fetch_line()` (on line 4). The standard library's *stdio.h* header file is **`#include`**`d` on line 3 to make the **`FILE`** symbol available for use in the next line. You will find an explanation of the *#include guards* **`#ifndef`** `...`, etc., on page 51.

The contents of the implementation file *fetch-line.c* are shown in an outline form in Listing 9.3. It defines two functions, `trim_line()` and `fetch_line()`, whose purposes I will now describe.

9.5.1 ▪ The function `trim_line()`

The function `trim_line()` that appears on line 5 in Listing 9.3 is declared **`static`**[29], indicating that it is only for internal use within the file *fetch-line.c*. The function's sole argument is a pointer `s` to a buffer which is expected to contain a properly `NUL`-terminated string. It trims the string of any leading or trailing whitespace and comments, as detailed

[28] You may feed any text file to the program; there is nothing special about *fgets-demo.c*, but it works nicely for demonstrating some of the features of our *fetch-line* module.

[29] See Section 1.6 regarding the **`static`** declaration specifier.

Listing 9.3: An outline of the file *fetch-line.c*, which is *fetch-line* module's implementation.

```
1 #include <string.h>
2 #include <ctype.h>
3 #include <stdlib.h>
4 #include "fetch-line.h"
5 ▶ static char *trim_line(char *s) ...
6 ▶ char *fetch_line(char *buf,
7                   int buflen, FILE *stream, int *lineno) ...
```

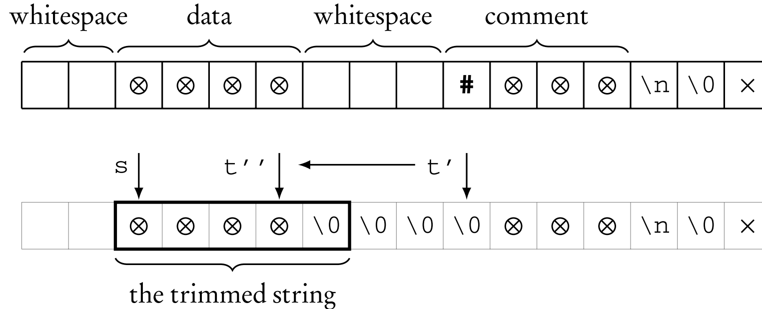


Figure 9.1: The top drawing shows a string buffer that contains a data segment along with leading and trailing whitespace and comments. The bottom drawing shows the associated trimmed string. A ⊗ indicates any character other than a `\0`, and a × indicates any character at all. The pointer `s` points to the first character of the trimmed string in a dark outline. The pointer `t` starts at `s` and slides down to either `\#` or `\0`, whichever comes first, and arrives at the position `t'`. Then in a retrograde motion it overwrites all whitespace by `\0` until it arrives at a nonwhitespace character at the position `t''`.

in Section 9.3, and returns a pointer to the trimmed string's first character. To help you get started, I have shown its operation graphically in Figure 9.1. The top part of Figure 9.1 shows a buffer that contains a data part along with leading and trailing whitespace characters and a comment. The bottom part shows the trimmed buffer in a dark outline. The arrows show pointer positions at the various stages of the processing of the string according to the following procedure:

1. Move the pointer `s` forward, one character at a time, skipping over whitespaces, if any, and stopping at the first nonwhitespace character. This ends the trimming of the data segment's *leading whitespace*. Now the pointer `s` is stationed at the start of the data segment. It is shown with the arrow marked `s` in Figure 9.1.

2. Set a second pointer, `t`, initially at `s`, and then move it forward until you arrive at either a `\0` or a **#** character. If it's **#**, change it to `\0` to truncate the comment as per the instructions in subsection 9.3.1. The stopping position of `t` is shown with the arrow marked `t'` in Figure 9.1 (but it will be just a plain `t` in your code).

3. If the position `t'` is other than `s`, move `t` backward, one character at a time, overwriting any whitespace with a `\0`, stopping at the first nonwhitespace character. This trims the data segment's *trailing whitespace*, if any. Now the pointer is stationed at the position marked with the arrow `t''` in Figure 9.1 (but it will be just a plain `t` in your code).

Listing 9.4: The function `fetch_line()` reads and returns the next nonempty trimmed line from the input stream. If the input line is too long to fit in the given buffer, it prints a diagnostic and calls `exit()`.

```
1  char *fetch_line(char *buf, int buflen, FILE *stream, int *lineno)
2  {
3      char *s;
4      if (fgets(buf, buflen, stream) == NULL)
5          return NULL;
6      ++*lineno;
7      if (buf[strlen(buf) - 1] ≠ '\n') {
8          fprintf(stderr, "*** reading error: input line %d too "
9                  "long for %s's buf[%d]\n",
10                 *lineno, __func__, buflen);
11         exit(EXIT_FAILURE);
12     }
13     s = trim_line(buf);
14     if (*s ≠ '\0')
15         return s;
16     else
17         return fetch_line(buf, buflen, stream, lineno); // recurse
18 }
```

At the completion of the process, the pointer `s` points to the trimmed data string. The function `trim_line()` returns that pointer to the caller. You have all the necessary information now to implement your own `trim_line()`.

Remark 9.2. If a string contains nothing but comments and whitespace, then it is reduced to an empty string after trimming. Consequently, the pointer returned by `trim_line()` points to `\0`. We use this observation in the next subsection to skip over empty lines in an input stream.

9.5.2 ▪ The function `fetch_line()`

The function `fetch_line()` that appears on line 6 in Listing 9.3 is presented in its entirety in Listing 9.4. The caller of `fetch_line()` supplies the buffer `buf` of length `buflen` and a `stream` which is open for reading. `fetch_line()` calls `fgets()` to read a line from `stream` into `buf` and then passes that line to `trim_line()` to strip it of comments and leading and trailing whitespace. If the trimmed string is nonempty, then `fetch_line()` returns a pointer to the start of that string; otherwise it calls itself recursively in search of a nonempty line. If the end of the input is reached, or if the call to `fgets()` experiences an error, `fetch_line()` returns `NULL`.

Our `fetch_line()` is not designed to resize the buffer. If an input line is too long to fit, `fetch_line()` takes a *very drastic action:* it prints a diagnostic message to `stderr` and then calls `exit()` to abort the program! There are other ways of handling such contingencies; I have chosen this extremely drastic route because it's the only useful action for our purposes.

How does `fetch_line()` tell if a line is too long to fit? Since the lines are read by `fgets()`, and since `fgets()` includes the line's terminating newline character in the fetched buffer, as shown in the "case 3" figure on page 64, then a properly fitting line will be a string terminated by the two characters `\n\0`; that is, the string's last character,

Listing 9.5: The file *fetch-line-demo.c* is a driver for the *fetch-line* module.

```c
#include <stdio.h>
#include "fetch-line.h"
#define BUFLEN 40
int main(void)
{
    char buf[BUFLEN];
    char *s;
    int lineno = 0;
    while ((s = fetch_line(buf, BUFLEN, stdin, &lineno)) != NULL)
        printf("trimmed line %3d: %s\n", lineno, s);
    return 0;
}
```

just before the null terminator, will be `\n`. In Listing 9.4 we determine a string's last character with the help of the standard library's `strlen()` function, which returns a string's length.

`fetch_line()`'s fourth argument, `lineno`, is a pointer to an integer that holds the value of the current line number. Whenever `fetch_line()` reads a line, it increments the number stored there. The caller may refer to that number when issuing diagnostics about possible errors in the input.

9.6 ▪ *Project fetch_line*

Part 9.1. Listing 9.5 shows a minimalistic driver to demonstrate the functionality of our *fetch-line* module. It sets up a buffer of length `BUFLEN` and then calls `fetch_line()` in a loop to read lines from the program's `stdin` into that buffer. It also sets up an integer counter, `lineno`, initialized to zero, whose address is passed to `fetch_line()`. As we saw in subsection 9.5.2, `fetch_line()` increments that counter every time it reads a line. Therefore `lineno` will contain the line number of the current line at any time. The program writes out that information to the `stdout` in Listing 9.5, line 10.

Encode, compile, and test your `fetch_line` program. See the sample session on page 67.

Part 9.2. Suppose `BUFLEN` is 40, as before. Create a file with gradually increasing line lengths, e.g.,

```
35xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
36xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
37xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
... more lines here ...
42xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

where the number at the beginning of each line equals the total length of that line. What happens when you feed that file to your program? Explain what you observe.

Part 9.3. Add whitespace (spaces, tabs, empty lines) and comments in various places in the file that you created in Part 9.2. Verify that your program behaves as intended.

Chapter 10

Generating random numbers

Prerequisites: Chapters 7, 8

10.1 ▪ The `rand()` and `srand()` functions

The C standard library provides a pseudorandom number generator through the `rand()` function. Together with the associated `srand()` function, they are declared in `stdlib.h` as "`int rand(void);`" and "`void srand(unsigned int seed);`".

Each call to `rand()` returns a random integer n with a uniform probability distribution in the range $0 \le n \le$ RAND_MAX. The value of RAND_MAX, a large positive integer, is built into your C library and cannot be changed. The C library on my computer has RAND_MAX $= 2^{31} - 1 = 2,147,483,647$. It may be something else on yours.[30] Here is a complete program that prints four random numbers:

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    for (int i = 0; i < 4; i++)
        printf("%d   ", rand());
    putchar('\n');
    return 0;
}
```

Here is what it prints:

```
1804289383   846930886   1681692777   1714636915.
```

If you run the program once again, it will print exactly the same four numbers! This is not a bug—even a statistician likes reproducible experiments!

Nevertheless, if you truly want your program to generate a different sequence of random numbers, invoke `srand(n)` with some positive integer n—called the random generator's *seed*—to alter the random generator's initial state, as in

```
#include <stdio.h>
#include <stdlib.h>
```

[30] You can find out the value of RAND_MAX on your machine by printing it in a C program, as in `printf("RAND_MAX = %u\n", RAND_MAX);` Alternatively, you may just look it up in *stdlib.h* if you know where to find that header file. On a typical Unix installation it is */usr/include/stdlib.h*.

```
int main(void)
{
    srand(7);                    // set seed to 7
    for (int i = 0; i < 4; i++)
        printf("%d   ", rand());
    putchar('\n');
    return 0;
}
```

Now the program prints

```
1045618677   1863967299   1272579899   461085871.
```

Setting the seed to 1 is equivalent to not calling `srand()` at all.

To generate random fractional numbers r in the closed interval $[0,1]$, divide the value returned by `rand()` by `RAND_MAX`, as in `r = (double)rand()/RAND_MAX`. The cast is necessary; otherwise the quotient is zero according to C's integer division rules. To generate random fractional numbers R in a prescribed interval $[a,b]$, generate them in $[0,1]$ as above and then map them to $[a,b]$ through scaling, as in $R = a + (b-a)r$.

Producing random *integers* on a prescribed integer range is a more challenging problem. To simulate a coin tossing experiment, for instance, we want random numbers in the set $\{0,1\}$. To simulate the tossing of dice, we want random numbers in the set $\{1,2,\dots,6\}$. Unfortunately there is no simple and perfect way of doing this. A reasonably good way of generating uniformly distributed random integers r in the set $\{0,1,\dots,n-1\}$ is through the formula

$$\mathtt{r} = \frac{\mathtt{rand()}}{\dfrac{\mathtt{RAND_MAX}}{\mathtt{n}} + 1} \qquad (\mathtt{n} \ll \mathtt{RAND_MAX}),$$

where both divisions are performed in the sense of integer arithmetic; that is, the quotient is computed and the remainder is discarded. (C's integer arithmetic does that by default.) The formula works well when n is *much smaller* than `RAND_MAX`.[31] To dramatize that statement, let us say `RAND_MAX` is 1,000. (This is far from realistic; on a typical implementation, `RAND_MAX` will be millions of times larger.) If $n = 6$, as in a die throwing experiment, then the likelihood of obtaining a 0 (or a 1, or 2, or 3, or 4) according to that formula is 167/1001, but the likelihood of obtaining a 5 is 166/1001, which is just slightly lower but not too bad. On the other hand, if $n = 30$, then the likelihood of obtaining a 0 (or a 1, or ...28) is 34/1001, but the likelihood of obtaining a 29 is 15/1001, which is significantly lower.

In the applications of interest to us, the values of n are millions of times smaller than `RAND_MAX`; therefore we run no risk of skewing our statistics by using that formula. In fact, we package the formula in a convenient function, which we call `random()`, for future use. Listing 10.1 shows the essentially one-line implementation.

Remark 10.1. The `inline` specifier in the function's declaration makes it an *inline function*, which has the effect of suggesting to the compiler that the calls to this function are to be as fast as possible. The C standard does not prescribe how that is to be accomplished. The compiler may, at its discretion, dismantle the function and conceptually embed its body where it's called, thus obviating the overhead of a function call altogether. Or it

[31] I learned about this formula from <http://www.c-faq.com/lib/randrange.html>. That page also offers a method, at the cost of some loss of efficiency, for dealing with situations where n is not necessarily small compared to `RAND_MAX`.

Listing 10.1: Returns a random integer in the set $\{0, 1, \ldots, n-1\}$ with uniform distribution. Assumes n is much smaller than RAND_MAX.

```
1 inline int random(int n)
2 {
3     return rand()/(RAND_MAX/n + 1);
4 }
```

may take some other measures to speed up the function's use. Or it may just ignore the inlining request altogether.

The `inline` specifier was introduced in C99. If your compiler does not support it, it's safe to remove it.

Remark 10.2. The C standard does not prescribe an algorithm for the pseudorandom number generator (PRNG) associated with `rand()`. Its only requirement is that RAND_MAX be at least 32767. The algorithm is left up to the implementation.

Earlier implementations of the standard library's `rand()` developed a (well-deserved) reputation for being too naive; the data they generated could hardly pass even the simplest tests for randomness. Serious users shunned the standard library's `rand()` and used PRNGs of their own, or something else made by third parties. Some computer platforms supplied their own (nonstandard) PRNG, usually called `random()`, as an alternative to the standard `rand()`. Things have improved over the years. The `rand()` and `random()` functions that comes with modern Linux distributions, for instance, both use the same rather sophisticated algorithm, which has a period of approximately $16 \times (2^{31} - 1)$. To learn more about PRNGs, consult *Numerical Recipes in C* [53], which devotes an entire chapter to the subject.

Remark 10.3. The name "`random`" used for the function defined in Listing 10.1 is quite legitimate within standard C. It is likely, however, that your computer platform supplies a nonstandard PRNG called `random()`, as noted in the previous paragraph. So, which of the two `random()` functions will your program use?

To avoid a potential clash between the two, be sure to tell your compiler to treat your program as strictly standard C. For example, if you use the GNU C compiler, invoke it with the `-std=c89` or the `-std=c99` flag. In the worst case, if you have a broken compiler that refuses to understand about standards, then rename the `random()` in Listing 10.1 to `Random()`, or somesuch.

10.2 ▪ Bitmap images

We intend to exercise our `random()` function of Listing 10.1 to produce *bitmap images* like those shown in Figure 10.1. A bitmap image is a rectangular array of black and white dots.

It is natural to represent a bitmap image as a matrix of zeros and ones. Each of the matrix's entries corresponds to a dot in the image. A zero represents a white dot, a 1 a black dot. The *Portable Bitmap* (PBM) convention for storing bitmap images formalizes this idea: You put the matrix of zeros and ones in a file, say *sample-bitmap.pbm*, and add a header consisting of the two-character *magic number*[32] `P1`, along with the image's width and the height (in that order). Then you may view the image by passing the file name

[32]See footnote 43 on page 115 for the meaning of the "magic number".

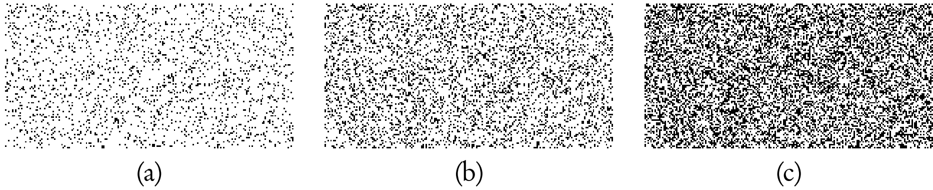


Figure 10.1: Three 100×200 bitmap images of random noise. From left to right, the fill ratio is 0.10, 0.20, and 0.40.

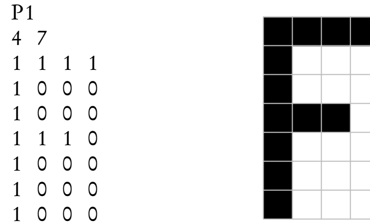


Figure 10.2: On the left are the contents of a PBM definition file. On the right is the corresponding 7×4 bitmap image. I have superimposed a grid to delineate the pixels. Beware that the image shown is *greatly magnified*; a 7×4 bitmap image in its natural size is too small for viewing on a computer screen, but you can ask your image viewer to magnify it.

sample-bitmap.pbm to almost any image viewer. Figure 10.2 shows the contents of a PBM file and the corresponding image.

Remark 10.4. As simple as it is, the plain PBM format which I have described here is the *most wasteful* way of storing a bitmap image. Much better ways are available. See Chapter 14 for further discussion of bitmaps and other forms of representing images.

10.3 ▪ The program

In principle, one can encapsulate the `random()` function of Listing 10.1 in a module consisting of a pair of files *random.h* and *random.c*. I will not take that route, however, since the function is so simple—a one-liner, actually—that the ado about an interface and an implementation will blow things way out of proportion. I would rather just insert a copy of the four lines of Listing 10.1 into a program, wherever it's needed. I will take that approach in this chapter and elsewhere.

Here is this chapter's project. Take an $m \times n$ matrix, and imagine that its entries represent a rectangular array of m rows and n columns of white dots. Then begin selecting random dots and coloring them black. Repeat until a certain percentage, say 10%, of the dots have been colored. That will produce a picture like Figure 10.1(a). We say that the *fill ratio* of that picture is 0.10. The pictures (b) and (c) have fill ratios of 0.20 and 0.40, respectively.

Figure 10.1(a) was produced by executing the command

```
$ ./random-pbm 100 200 1 0.10 rand-10.pbm
```

Those command-line arguments request a 100×200 image and set the random number generator's seed to 1, the fill ratio to 0.10, and the name of the output file (that is, the image

Listing 10.2: An outline of the file *random-pbm.c*.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include "array.h"
4 ▶static inline int random(int n) ...
5 ▶static int write_pbm(char **M, int m, int n, char *outfile) ...
6 ▶static char **make_random_matrix(int m, int n, double f) ...
7 ▶static void show_usage(char *progname) ...
8 ▶int main(int argc, char **argv) ...
```

file) to *rand-10.pbm*. If the program is invoked with insufficient or illegal command-line arguments, it prints a help message to the `stderr` and exits, as in

```
$ ./random-pbm
Usage:  ./random-pbm m n s f outfile
    writes an mxn random bitmap to a file named 'outfile'
    f: fill ratio  0.0 <= f <= 1.0
    s: integer >= 1: seeds the random number generator
```

We use Chapter 7's *Xmalloc* module and Chapter 8's *array.h* header file to allocate memory for the bitmap's matrix. Therefore, following the suggestions of Chapters 2 and 6, the program's directory will look like this:

```
$ cd random
$ ls -F
Makefile  array.h@  random-pbm.c  xmalloc.c@  xmalloc.h@
```

I will describe the contents of the file *random-pbm.c* in the next section.

10.4 ▪ The file *random-pbm.c*

Listing 10.2 gives an outline of the file *random-pbm.c*. The function `random()` that appears on line 4 is the same as that shown in Listing 10.1. The function `show_usage()` that appears on line 7 is responsible for printing the help message shown in Section 10.3. I will let you write it. The remaining three functions are described in the following subsections.

10.4.1 ▪ The function `main()`

The function `main()` that appears on line 8 of Listing 10.2 is responsible for parsing the command-line arguments and then directing the rest of the program's flow. The initial part of the function `main()` is shown in Listing 10.3. Let me explain some of the details.

Line 5. The matrix `M` will hold the bitmap's data. Since the data consists of zeros and ones, any integer data type will do. We choose **`char`** here since it is the smallest such data type.

Line 8. The variable `status` holds the program's exit status. We initialize it to the default of `EXIT_FAILURE`. Later on, after obtaining reassurance that the program has succeeded (line 38 in Listing 10.4), we will change it to `EXIT_SUCCESS`.

Line 9. As we saw in Section 10.3, the program is expected to be invoked with five arguments, and therefore `argc` should be 6; see Section 4.7 regarding command-line

Listing 10.3: The initial part of the function `main()` in file *random-pbm.c*.

```
1  int main(int argc, char **argv)
2  {
3      int m, n, s;       // image is m × n, seed is s
4      double f;          // fill ratio
5      char **M;
6      char *outfile;
7      char *endptr;
8      int status = EXIT_FAILURE;
9      if (argc ≠ 6) {
10         show_usage(argv[0]);
11         return EXIT_FAILURE;
12     }
13     m = strtol(argv[1], &endptr, 10);
14     if (*endptr ≠ '\0' || m < 1) {
15         show_usage(argv[0]);
16         return status;
17     }
18      // ··· similarly, extract n, s, f, and outfile
```

arguments. If `argc` is not 6, we print a usage message and exit. We pass `argv[0]` to `show_usage()` so that it can print the program's name along with the message. See the sample message shown in Section 10.3.

Line 13. We call the standard library's `strtol()` function (Chapter 5) to translate the string `argv[1]` to an integer. Subsequently, we verify that `argv[1]` is a proper representation of an integer and that its numerical value is no less than 1. If not, we print a usage message and exit.

Lines 18–34. Here I have excised several lines from `main()` to give you an opportunity to insert your own code. You will need to extract the integers `n` and `s`, the floating point number `f`, and the string `outfile` from the command-line arguments. You will use the `strtod()` function (Chapter 5) to extract the value of `f`.

The remaining part of the function `main()` is shown in Listing 10.4. Let us examine the details.

Line 35. Now that we have extracted the user-specified value of the seed `s` for the random number generator, we call `srand()` to set the seed to `s`.

Line 36. We call the function `make_random_matrix()` (to be described later) to produce an $m \times n$ matrix of random zeros and ones with a fill ratio of f.

Line 37. We call the function `write_pbm()` (to be described later) to write the matrix's data as a PBM file. The function returns 1 on success and 0 on failure. (The most likely cause of failure is the lack of permission to open the file for writing.) If the call returns 1, then it has succeeded; therefore we change the value of `status` to `EXIT_SUCCESS`.

Listing 10.4: The end of the function `main()` in file *random-pbm.c*.

```
35      srand(s);
36      M = make_random_matrix(m, n, f);
37      if (write_pbm(M, m, n, outfile) == 1)
38          status = EXIT_SUCCESS;
39      free_matrix(M);
40      return status;
41  }
```

Listing 10.5: The function `make_random_matrix()` in file *random-pbm.c*.

```
1   static char **make_random_matrix(int m, int n, double f)
2   {
3       char **M;
4       int i, j, k;
5       make_matrix(M, m, n);
6       for (i = 0; i < m; i++)
7           for (j = 0; j < n; j++)
8               M[i][j] = 0;
9       k = 0;
10      while (k < f*m*n) {
11          i = random(m);
12          j = random(n);
13          if (M[i][j] == 0) {
14              M[i][j] = 1;
15              k++;
16          }
17      }
18      return M;
19  }
```

10.4.2 ▪ The function `make_random_matrix()`

The function `make_random_matrix()` that appears on line 6 of Listing 10.2 allocates memory for an $m \times n$ matrix, fills it with random zeros and ones (with a fill ratio of f), and returns a pointer to that matrix. Listing 10.5 shows the function in its entirety. The implementation is mostly self-explanatory; therefore I will highlight the interesting aspects only. First, we call `make_matrix()` (defined in *array.h*) to allocate memory for the matrix, and then we zero all its entries. Recall that zero in the PBM specification means white.

In the **`while`**-loop that follows, we call `random()` to generate random row and column indices `i` and `j` in the ranges `0 ≤ i < m` and `0 ≤ j < n`. If `M[i][j]` is zero, we change it to one and then increment the counter `k`, which keeps a count of the black dots. The test is necessary since if `M[i][j]` is already black, making it black again has no effect.

10.4.3 ▪ The function `write_pbm()`

The function `write_pbm()` that appears on line 5 of Listing 10.2 receives the bitmap matrix created by `make_random_matrix()` and writes it in the PBM format to a file whose name it receives as an argument. It returns 1 on success and `0` on failure.

I will describe the function's overall plan in words and let you fill in the details. First, call the standard library's `fopen()` to open the output file for writing. If unsuccessful, print an appropriate message to the `stderr`, and then return 0. Next, write the image's header as described in Section 10.2. See Figure 10.2 for a sample. Then write the data part according to the contents of the matrix `M`. Finally, close the output stream and return 1. You may use the `print_matrix()` macro from Part 8.3 of *Project array.h* (page 60) to print the matrix `M`.

10.5 ▪ *Project Random Bitmaps*

Complete the file *random-pbm.c* according to the given instructions, and then compile and experiment. Width and height values in the range 100 to 400 produce bitmaps of reasonable size for viewing on a computer screen.

There are very many utilities for viewing images. The choice depends in your platform and personal preference. I do all of my work on a Linux laptop. Mostly I use `feh` for viewing images. Occasionally I use the `display` utility that comes with the *ImageMagick* package and the venerable `xv`, which, unfortunately, is no longer actively maintained.

Chapter 11
Storing sparse matrices

Prerequisites: *Chapters 7, 8*

11.1 ▪ Introduction

A *sparse matrix* is a matrix whose entries are mostly zero. Large sparse matrices occur quite frequently in computing. In solving a linear Partial Differential Equation (PDE) with the Finite Element Method (FEM), for instance, one subdivides the equation's domain into a fine mesh of simple objects such as triangles or tetrahedra and approximates the PDE's solution with low degree polynomials in each. This reduces the task of solving the PDE to solving a linear system of equations $Ax = b$, where the $n \times n$ coefficient matrix A tends to be quite large; $n = 10{,}000$ is not out of the ordinary. Storing such a humongous matrix can pose challenges for the computer's memory and disk utilization. Fortunately, it is hardly ever necessary to store the matrix in toto since by the nature of the FEM, all but a tiny fraction of the matrix's entries are zeros in general.

Depending on the FEM's algorithm, the total number of nonzero entries in the $n \times n$ matrix A may be as little as $3n$. When $n = 10{,}000$, for instance, such a matrix will have a total of $100{,}000{,}000$ entries, only $30{,}000$ of which are nonzero. In other words, only one out of 3,333 entries is nonzero! So it would make sense to store only the nonzero entries along with their indices into A, rather than a hundred million zeros.

There is no standard way of storing sparse matrices—see [18, Chapter 2] and [6, Section 4.3] for surveys of the various methods—although the *Compressed Column Storage* (CCS) format appears to be the favorite choice nowadays, perhaps because that's what MATLAB uses.

It is this chapter's objective to introduce the CCS format and see how it translates to C code. The functions `sparse_pack()` and `sparse_unpack()` developed in the process convert a matrix to and from the CCS format. I must point out that we have no practical use for these functions in the rest of the book since our work with sparse matrices will be handled through the facilities provided by the UMFPACK library, which is the subject of the next chapter. Interacting with UMFPACK, however, does require some understanding of the CCS format, and that's what you are expected to take with you from this chapter.

11.2 ▪ The CCS format

I will explain the CCS format in terms of the concrete 4×5 matrix

$$A = \begin{pmatrix} 0 & 7 & 0 & 0 & 1 \\ 0 & 4 & 0 & 3 & 0 \\ 6 & 6 & 5 & 1 & 4 \\ 5 & 5 & 0 & 0 & 0 \end{pmatrix},$$

which is neither large nor truly sparse; nevertheless, it will serve quite adequately for conveying the idea behind CCS.

The matrix has 11 nonzero entries. Following [16] and [18], I will write `nz` for the number of a matrix's nonzero entries. The obvious—but not the best—way of storing the matrix's nonzero entries is as a set of `nz` triplets (i, j, A_{ij}) of locations and values. We put the first components of such triplets into a vector `Ai`, the second components into a vector `Aj`, and the third components into a vector `Ax`, and thus we get three vectors of length `nz` each:

$$\begin{aligned} \texttt{Ai} &= [\,2 \quad 3 \quad 0 \quad 1 \quad 2 \quad 3 \quad 2 \quad 1 \quad 2 \quad 0 \quad 2\,], \\ \texttt{Aj} &= [\,0 \quad 0 \quad 1 \quad 1 \quad 1 \quad 1 \quad 2 \quad 3 \quad 3 \quad 4 \quad 4\,], \\ \texttt{Ax} &= [\,6 \quad 5 \quad 7 \quad 4 \quad 6 \quad 5 \quad 5 \quad 3 \quad 1 \quad 1 \quad 4\,]. \end{aligned}$$

(Note that we are counting the row and column indices beginning with zero.) To produce this list, I scanned the matrix by columns from top to bottom and then from left to right. That's where the "column" in the CCS comes in.[33] There is a "Fortran accent" in this since that's the order in which matrix entries are stored in Fortran. (In contrast, C's built-in array types store two-dimensional arrays in the row order.) The CCS format was developed when Fortran was—some say it still is—the dominant programming language in scientific computing.

As I noted above, the storage of A as the three vector set { `Ai`, `Aj`, `Ax` } *is not* the best we can do. A hint to the suboptimality of that representation is provided by the repetitive nature of the entries of the `Aj` vector. Have a look. The first two "0"s in `Aj` tell us that the matrix A has two nonzero entries in its first column. The next four "1"s in `Aj` tell us that A has four nonzero entries in its second column, and so on. Can't we express that information in a more concise way? Yes, we can! For instance, the vector

$$\texttt{A'} = [\;2 \quad 4 \quad 1 \quad 2 \quad 2\;]$$

conveys exactly the same information as `Aj` (two entries in the first column, four entries in the second column, etc.) and it is shorter.

The CCS format is a slight variant of this idea. For an $m \times n$ matrix A it introduces a vector `Ap` of length $n+1$ with the property that `Ap[j]` gives the index in the vector `Ax` of where the matrix's column `j` begins. More precisely, the vector `Ap` is such that the entries in the matrix's `j`th column are given by

$$\{\ \texttt{Ax[k]} \ :\ \texttt{Ap[j]} \le \texttt{k} < \texttt{Ap[j+1]}\ \}.^{34} \tag{11.1}$$

[33] It should be obvious that the order in which the matrix's entries are scanned is immaterial for the purpose of its representation as the { `Ai`, `Aj`, `Ax` } three-vector set. But the order *does matter* where the vector `Ap` is introduced later on.

[34] This implies, in particular, that `Ap[n] = nz`.

For our matrix A we have `Ap` = [0 2 6 7 9 11] because

the values	{ `Ax[k]` : 0 ≤ `k` < 2 }	go to the matrix's first column,
the values	{ `Ax[k]` : 2 ≤ `k` < 6 }	go to the matrix's second column,
the values	{ `Ax[k]` : 6 ≤ `k` < 7 }	go to the matrix's third column,
the values	{ `Ax[k]` : 7 ≤ `k` < 9 }	go to the matrix's fourth column,
the values	{ `Ax[k]` : 9 ≤ `k` < 11 }	go to the matrix's fifth column.

To conclude, the matrix A and its CCS representation are

$$A = \begin{pmatrix} 0 & 7 & 0 & 0 & 1 \\ 0 & 4 & 0 & 3 & 0 \\ 6 & 6 & 5 & 1 & 4 \\ 5 & 5 & 0 & 0 & 0 \end{pmatrix} \iff \begin{cases} \texttt{Ap} = [\,0 \quad 2 \quad 6 \quad 7 \quad 9 \quad 11\,], \\ \texttt{Ai} = [\,2 \quad 3 \quad 0 \quad 1 \quad 2 \quad 3 \quad 2 \quad 1 \quad 2 \quad 0 \quad 2\,], \\ \texttt{Ax} = [\,6 \quad 5 \quad 7 \quad 4 \quad 6 \quad 5 \quad 5 \quad 3 \quad 1 \quad 1 \quad 4\,]. \end{cases}$$

The sparse matrix representation is not particularly useful in this instance since it would have been more economical to store the matrix's $4 \times 5 = 20$ entries, zeros and all, instead of the $6 + 11 + 11 = 28$ numbers that make up the vectors `Ap`, `Ai`, `Ax`. The benefits of sparse storage become apparent only when the matrix is large and has a small fraction of nonzero entries.

11.3 ▪ The program

We are going to write a *sparse matrix module* in a file *sparse.c* that contains two C functions, `sparse_pack()` and `sparse_unpack()`. The first receives an $m \times n$ matrix and computes the corresponding CCS vectors `Ap`, `Ai`, `Ax`. The second does the reverse; that is, it receives the CCS vectors `Ap`, `Ai`, `Ax` and constructs the matrix A. Additionally, we will write a test/demo program in *sparse-test.c* to exercise those functions. The program depends on *array.h* (Chapter 8) and *xmalloc.[ch]* (Chapter 7) for allocating memory for vectors and matrices; therefore, following the suggestions of Chapters 2 and 6, your project's directory will look like this:

```
$ cd sparse
$ ls -F
Makefile array.h@  sparse.c  sparse.h  xmalloc.c@  xmalloc.h@
```

Our implementations of `sparse_pack()` and `sparse_unpack()` work with matrices of type **`double`**. These are the types of matrices that most frequently require sparse storage. Should you have a need to work with matrices of a different type, you will have to write another pair of functions for that type. It is possible to apply Chapter 8's preprocessor trick to produce type-generic versions of these, but it's probably not worth the effort since the need for sparse storage of matrices other than the type **`double`** is so rare.

11.4 ▪ The files *sparse.[ch]*

The functions `sparse_pack()` and `sparse_unpack()`, defined in the file *sparse.c*, deal with *preallocated vectors and matrices*. That is, the caller is expected to have allocated properly sized vectors and matrices before calling these function. They do no memory allocations on their own.

The implementation of the function `sparse_pack()` is quite straightforward, so I will leave it to you to write. The function `sparse_unpack()`, which is only slightly more involved, is shown in Listing 11.1. It receives the CCS vectors `Ap`, `Ai`, `Ax` and populates the $m \times n$ matrix A with them.

Listing 11.1: The implementation of the function `sparse_unpack()`. It receives the CCS vectors `Ap`, `Ai`, `Ax` and populates the $m \times n$ matrix A with them.

```
1  void sparse_unpack(double **a, int m, int n,
2          int *Ap, int *Ai, double *Ax)
3  {
4      int i, j, k;
5      for (i = 0; i < m; i++)
6          for (j = 0; j < n; j++)
7              a[i][j] = 0.0;
8
9      for (j = 0; j < n; j++)
10         for (k = Ap[j]; k < Ap[j+1]; k++) {
11             i = Ai[k];
12             a[i][j] = Ax[k];
13         }
14
15 }
```

The function begins with storing zeros for all entries in the matrix A. After all, A is supposedly sparse; therefore most of its entries are zeros anyway. The nested **`for`**-loop that begins on line 9 decodes the CCS vectors. The outer **`for`**-loop steps through the columns $0 \leq j < n$. The inner **`for`**-loop extracts that column's entries according to the formula (11.1) on page 80 and inserts them into the matrix A.

To complete the module, write a header file, *sparse.h*, that provides the module's interface. Don't forget to put *#include guards* in *sparse.h* and **`#include`**`"sparse.h"` in *sparse.c*.

11.5 ▪ *Project Sparse Matrix*

Complete the files *sparse.c* and *sparse.h*, as instructed in the previous sections. Then write file *sparse-test.c*, which provides a test/demo of our *sparse* module. I will describe its contents in words and let you make it into a C program.

- Create the 4×5 matrix A of Section 11.2. Thus, let $m = 4$, $n = 5$, and then

  ```
  double **a;
  make_matrix(a, m, n);
  a[0][0] = 0; a[0][1] = 7; ...
  ```

- Step through the matrix and count the number of its nonzero entries. Call it `nz`.
- Print the matrix and `nz`.
- Call `make_vector()` to make the CCS vectors `Ap`, `Ai`, `Ax` of appropriate lengths.
- Call `sparse_pack()` to encode the matrix into the CCS vectors.
- Print the vectors `Ap`, `Ai`, `Ax`.
- Make an $m \times n$ matrix `b`. Call `sparse_unpack()` to unpack the CCS vectors into the matrix `b`.

- If your code is working properly, the matrices `a` and `b` should be identical; therefore their difference should be zero. To verify this, make a third $m \times n$ matrix; let's call it `c`. In a doubly nested **for**-loop, set `c[i][j] = a[i][j] - b[i][j]`. Print the matrix `c`. Is it all zeros?
- Free all allocated memory before exiting.

For your reference, here is what my program prints:

```
The sparse matrix is
       0         7         0         0         1
       0         4         0         3         0
       6         6         5         1         4
       5         5         0         0         0
matrix is 4x5, nz = 11

Ap =    0    2    6    7    9   11
Ai =    2    3    0    1    2    3    2    1    2    0    2
Ax =    6    5    7    4    6    5    5    3    1    1    4

The difference of the original and reconstructed matrices:
       0         0         0         0         0
       0         0         0         0         0
       0         0         0         0         0
       0         0         0         0         0
```

As you see, there are quite a few calls to print vectors and matrices. You may put the type-generic `print_vector()` and `print_matrix()` macros developed in Part 8.3 of *Project array.h* (page 60) to good use here.

Chapter 12

Sparse systems: The UMFPACK library

Prerequisites: *Chapters 7, 8, 11*

12.1 ▪ Introduction

UMFPACK (pronounced in two syllables: "umph"–"pack") is a library of C functions for solving systems of linear equations $Ax = b$, where A is an $n \times n$ matrix and b is an n-vector. Although it will work with any nonsingular A, it is particularly effective in the situations where A is large and sparse. The library, written by Tim Davis, is available as an open and free software from

`<http://www.cise.ufl.edu/research/sparse/umfpack/>`.

It incorporates state-of-the-art algorithms for solving large sparse systems developed by Davis and a host of collaborators, and underlies many computational algorithms in software packages such as Comsol, Maple, and MATLAB.

This chapter introduces some of the most basic aspects of the UMFPACK library and shows how to interface with it in a C program. We cover only what is needed in applications to finite elements in Chapters 25 and 26. To learn more, you should read UMFPACK's *Quick Start Guide* and *User Guide* (17 pages and 133 pages, respectively, as of Version 5.4.0, dated May 20, 2009) that come with UMFPACK as PDF files. Additionally, you may benefit from reading Davis's book [16] and the articles [14, 15].

12.2 ▪ The basics

UMFPACK expects the $n \times n$ coefficient matrix A of the system $Ax = b$ to be given in the CCS format, that is, as the three vectors `Ap`, `Ai`, and `Ax`, as seen in Chapter 11. The vector `Ap` is of length $n+1$, while `Ai` and `Ax` are of length `nz` each, where `nz` is the count of A's nonzero entries. The system's right-hand side, b, is an ordinary vector of length n. Solving the system $Ax = b$ proceeds in three stages:

Stage 1: Symbolic analysis. We pass the matrix A (actually, the CCS vectors `Ap`, `Ai`, and `Ax`) to the UMFPACK function `umfpack_di_symbolic()` for *symbolic analysis*. The goal of this stage is to analyze the *pattern* of distribution of the nonzero entries in the sparse matrix A; their numeric values are immaterial. UMFPACK applies a series of row and column permutations to arrange the pattern of the nonzeros in a certain optimal way. This transforms the matrix A to a matrix PAQ, where P and Q are *permutation matrices*.

Remark 12.1. A permutation matrix is the result of shuffling the rows or columns of the $n \times n$ identity matrix. If the permutation matrix P is produced by shuffling the rows of the identity matrix, then the left-multiplication PA with an arbitrary $n \times n$ matrix A applies exactly those row shuffles to A. If the permutation matrix P is produced by shuffling the columns of the identity matrix, then the right-multiplication AP with an arbitrary $n \times n$ matrix A applies exactly those column shuffles to A. A permutation matrix is an *orthogonal matrix*, that is, $P^T P = I$, where I is the identity matrix and P^T is the transpose of P.

Stage 2: Numeric analysis. We pass the matrix A and the permutation data (effectively the P and Q permutations matrices from the previous step) to the UMFPACK function `umfpack_di_numeric()`, which applies *Gaussian elimination* to perform an *LU factorization* of the permuted matrix; that is, it computes L and U so that $PAQ = LU$, where L and U are $n \times n$ matrices, L is lower-triangular, and U is upper-triangular.

Stage 3: Solving the system. We pass the matrix A, the LU factorization data from the previous step, the vector b, and the address of a preallocated vector x to the UMFPACK function `umfpack_di_solve()`, which solves the system $Ax = b$ and places the solution in the vector x.

Let me explain that last step in a little more detail. From the LU factorization $PAQ = LU$ and the orthogonality of P and Q it follows that $A = P^T LUQ^T$; therefore the system $Ax = b$ takes the form $P^T LUQ^T x = b$, or equivalently, $LUQ^T x = Pb$. Introducing the temporary variable $y = UQ^T x$ reduces the equation to the linear system $Ly = Pb$, which is quite trivial to solve for y since the coefficient matrix L is lower-triangular. Once we have y, we go back to $y = UQ^T x$ and rewrite it as $y = Uz$ by introducing the temporary variable $z = Q^T x$. The system of linear equations $Uz = y$ is trivial to solve for z since the coefficient matrix U is upper-triangular. Once we have z, we compute the solution x of the original system from $z = Q^T x$, that is, $x = Qz$.

That's a lot of computation, but fortunately we don't have to do it ourselves. UMFPACK sees to it that it's done. For our part, we need to know how to invoke the three steps that lead to the solution. I will explain those through several sample programs in the rest of this chapter.

12.3 ▪ The program

We are going to write several stand-alone demos, *umfpack-demo[123].c*, which show, in successively greater details, how to work with the UMFPACK library. The programs rely on *xmalloc.[ch]* (Chapter 7) and the header file *array.h* (Chapter 8) for allocating and freeing memory for vectors and matrices. Furthermore, we will have use for the *sparse* module of Chapter 11. Therefore, following the suggestions of Chapters 2 and 6, your project's directory will look like this:

```
$ cd umfpack
$ ls -F
Makefile  sparse.c@  umfpack-demo1.c  umfpack-demo3.c  xmalloc.h@
array.h@  sparse.h@  umfpack-demo2.c  xmalloc.c@
```

To compile and run this chapter's programs, you will need a properly installed UMFPACK library on your computer. On most Linux distributions, UMFPACK is supplied as an optional package which may be installed with a few commands or mouse clicks.

The package may be called `umfpack-dev`, `libsuitesparse-dev`, or somesuch, depending on the distribution. On a Mac, you may download and install UMFPACK from `<http://www.macports.org/>`. On other operating systems you may have to download UMFPACK's source and then compile and install the library yourself. UMFPACK's *User Guide* tells you how.

The header file *umfpack.h* provides the interface to the UMFPACK library. In many installations, that header file and several dozen others that come with UMFPACK are placed into a directory of their own, as a subdirectory of the main "include" directory which is in the compiler's default search path. On my computer that directory is named *suitesparse*; therefore, to include *umfpack.h* in my programs, I do

```
#include <suitesparse/umfpack.h>
```

Installations vary. You will have to find out where your *umfpack.h* lives and **`#include`** it accordingly.

Additionally, you will need to tell the compiler where to find the library itself (not the header file) for the purposes of linking. If the library is installed in the compiler's search path, then something like this should do:

```
$ gcc -Wall -std=c99 -pedantic -O2 umfpack-demo1.c -lumfpack
```

Again, there are too many variations on the possible names and locations of the library for me to provide a definitive recipe. You may ask your computer guru for help if the basic suggestions above don't work for you.

At any rate, see Section 6.9 on how to present the required information in your *Makefile*.

12.4 ▪ *umfpack-demo1.c*

The file *umfpack-demo1.c* presents an absolutely bare-bones demo of the UMFPACK library. It solves the system of five equations in five unknowns $Ax = b$, where

$$A = \begin{pmatrix} 2 & 3 & 0 & 0 & 0 \\ 3 & 0 & 4 & 0 & 6 \\ 0 & -1 & -3 & 2 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 4 & 2 & 0 & 1 \end{pmatrix}, \qquad b = \begin{pmatrix} 8 \\ 45 \\ -3 \\ 3 \\ 19 \end{pmatrix}.$$

You may verify that the solution to the system is $x = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \end{pmatrix}^T$. This problem comes directly from UMFPACK's *User Guide*.

The demo file *umfpack-demo1.c* consists of a single function, `main()`, outlined in Listing 12.1. Lines 1 through 15 essentially duplicate what was done in *sparse.c* in Chapter 11's *Project Sparse Matrix*; that is, they encode the matrix A into the CCS vectors `Ap`, `Ai`, and `Ax`.[35] The only new feature here are the variables declared on line 3, which will be explained below. Let us look at the rest of the `main()` now.

Line 16. The first call to the UMFPACK library occurs on line 16. The function `umfpack_di_symbolic()` corresponds to **Stage 1** of the three stages of UMFPACK's operation described in Section 12.2. The first two arguments are the number of rows and columns of the matrix A. Since our A is square, then both arguments

[35] In a small problem such as the present one, you may compute the CCS vectors by hand, in which case there won't be a need for Chapter 11's `sparse_pack()` function at all.

Listing 12.1: An outline of the function `main()` in *umfpack-demo1.c*.

```
1  int main(void)
2  {
3      void *Symbolic, *Numeric;
4      double **a;
5      double *b, *x, *Ax;;
6      int *Ap, *Ai;
7      int n = 5;
8      int i, j, nz;
9      ... allocate memory for the matrix a and the vectors b and x ...
10     ... populate the matrix a and vector b with the given numbers ...
11     ... compute nz as in Project Sparse Matrix ...
12     ... print out a, b, and nz as in Project Sparse Matrix ...
13     ... allocate memory for the vectors Ap, Ai, and Ax ...
14     sparse_pack(a, n, n, Ap, Ai, Ax);    // populate the CCS vectors
15     ... print out Ap, Ai, Ax ...
16     umfpack_di_symbolic(n, n, Ap, Ai, Ax, &Symbolic, NULL, NULL);
17     umfpack_di_numeric(Ap, Ai, Ax, Symbolic, &Numeric, NULL, NULL);
18     umfpack_di_solve(UMFPACK_A, Ap, Ai, Ax, x, b, Numeric, NULL, NULL);
19     ... print out x ...
20     umfpack_di_free_symbolic(&Symbolic);
21     umfpack_di_free_numeric(&Numeric);
22     ... also free the allocated memory for a, b, x, Ap, Ai, Ax ...
23     return 0;
24 }
```

are n. The next three arguments are the three vectors of A's CCS encoding, as computed earlier with the help of our function `sparse_pack()` on line 14. The next argument is the address of the `Symbolic` pointer variable which is declared on line 3. UMFPACK sets `Symbolic` to point to an (opaque) structure which it allocates and uses internally for the purpose of storing the permutation matrices P and Q (or rather, their encodings) described in Section 12.2. The final two arguments are `NULL`. It is possible to set them to certain control arrays to pass additional information to UMFPACK or extract information from it. We will not use those control arrays in this book.

Line 17. The call to `umfpack_di_numeric()` on line 17 corresponds to **Stage 2** of the three stages described in Section 12.2. All arguments except the fifth are those we have seen before. The fifth argument receives the address of the `Numeric` pointer variable which is declared on line 3. UMFPACK sets `Numeric` to point to an (opaque) structure which it allocates and uses internally for the purpose of storing information regarding the matrix's LU factorization.

Line 18. The call to `umfpack_di_solve()` on line 18 corresponds to **Stage 3** of the three stages described in Section 12.2. We have encountered all the arguments except the first. The first argument is set to the preprocessor symbol `UMFPACK_A` which tells UMFPACK to solve the system $Ax = b$ and place the computed solution in the vector `x`. That's all we need for the purposes of this book's projects, but you may be interested to know that you may solve the system $A^T x = b$ by setting the first argument to `UMFPACK_At`. UMFPACK's *User Guide* gives more than a dozen other options for the first argument.

Line 19. Here we print the solution x of the problem.

Lines 20 and 21. I have pointed out that UMFPACK stores permutation and factorization data in internal structures and sets the Symbolic and Numeric variables to point to them. It also provides functions umfpack_di_free_symbolic() and umfpack_di_free_numeric() to free the memory associated with those structures. Here we invoke those function to release the memory.

Line 22. We free the memory associated with the matrix a and several vectors.

You have all the necessary information for completing, compiling, and running *umfpack-demo1.c*. Do it now! Here is what my version prints:

```
$ ./umfpack-demo1
The sparse matrix is:
       2        3        0        0        0
       3        0        4        0        6
       0       -1       -3        2        0
       0        0        1        0        0
       0        4        2        0        1
matrix is 5x5, nz = 12

Ap =    0    2    5    9   10   12
Ai =    0    1    0    2    4    1    2    3    4    2    1    4
Ax =    2    3    3   -1    4    4   -3    1    2    2    6    1

x =    1    2    3    4    5
```

12.5 ▪ *umfpack-demo2.c*

Our next program, *umfpack-demo2.c*, is only a minor variation on *umfpack-demo1.c*; therefore you may copy your first program to *umfpack-demo2.c* and then make the necessary changes.

Most UMFPACK library functions return status codes to indicate the success or failure of the computations which they perform. In *umfpack-demo1.c* I ignored the return values in order to bring out the simple structure of the program's logic. But ignoring the returned status values is hardly advisable. Things *can* go wrong. Memory allocation may fail, the matrix may be singular or very ill-conditioned, etc. A well-written program should be prepared to take proper action when such exceptional conditions occur.

There are dozens of status codes defined in UMFPACK, the most important of which is UMFPACK_OK, which indicates a successful return from an UMFPACK function. Other common status codes are UMFPACK_ERROR_out_of_memory, which is returned by any function when it runs out of memory, and

UMFPACK_WARNING_singular_matrix,

which is returned by umfpack_di_numeric() if the *LU* factorization succeeds but it detects that the matrix is singular. There is nothing wrong with a matrix being singular as far as its *LU* factorization is concerned, but such a matrix is not suitable for passing to umfpack_di_solve() since it will amount to dividing by zero.

To account for the return values, we introduce a variable "**int** status" and then replace the line 16 in Listing 12.1 with

```
status =  umfpack_di_symbolic(n, n, Ap, Ai, Ax, &Symbolic, NULL, NULL);
if (status ≠ UMFPACK_OK) {
    fprintf(stderr, "file %s, line %d: umfpack_di_symbolic() failed\n",
            __FILE__, __LINE__);
    return EXIT_FAILURE;
}
```

Padding every occurrence of an UMFPACK function this way results in a lot of clutter. A better strategy is to isolate the error reporting to a function of its own. Here is an idea. Define

```
static void error_and_exit(int status, const char *file, int line)
{
    fprintf(stderr, "*** file %s, line %d: ", file, line);
    switch (status) {
        case UMFPACK_ERROR_out_of_memory:
            fprintf(stderr, "out of memory!\n");
            break;
        case UMFPACK_WARNING_singular_matrix:
            fprintf(stderr, "matrix is singular!\n");
            break;
        default:
            fprintf(stderr, "UMFPACK error code %d\n", status);
    }
    exit(EXIT_FAILURE);
}
```

It prints an informative error message to `stderr` and then calls `exit()` to terminate the program. With this function on hand, you may replace line 16 in Listing 12.1 with

```
status =  umfpack_di_symbolic(n, n, Ap, Ai, Ax, &Symbolic, NULL, NULL);
if (status ≠ UMFPACK_OK)
    error_and_exit(status, __FILE__, __LINE__);
```

This is more compact than the previous suggestion.

Change lines 17 and 18 in the same way, and then compile and test the program. The output should be identical to that of the previous version.

To check the proper functioning of `error_and_exit()`, change the matrix A to a singular one. (You may do that, for instance, by changing the two entries in the first column to zeros.) Try again. Does your program behave as designed?

12.6 ▪ *umfpack-demo3.c* and the triplet form

Each of the two previous demo programs begins with a fully formed matrix A and then calls `sparse_pack()` to produce its CCS representation vectors `Ap`, `Ai`, and `Ax`. If you think about it, this is totally insane! The sole purpose of sparse storage is to avoid wasting memory in storing a matrix's zero entries. But what do we do? In the first place, we commit the sin of storing the *entire matrix*, zeros and all, in the array A. Then we add insult to injury by allocating yet *more memory* for its CCS vectors. That can't be right, can it? No, it's not. A proper implementation will have *no matrix A at all!* It will begin with a list of A's *nonzero entries only* and go from there. Let's see how.

12.6.1 ▪ The triplet form

How do we produce a matrix's nonzero entries without producing the matrix itself? A nonzero entry may be identified as a triplet (i, j, x), where the i and j are the entry's row and column indices, and x is its value. A sparse matrix is specified completely by a list of such triplets. There is no need to store the full matrix at all.

Let's say we wish to store N such triplets. One way is to make a vector of length N whose entries are structures that hold triplets. Another way, and this is the way it's done in UMFPACK, is to make three vectors of length N each, let's call them `Ti`, `Tj`, `Tx`, and store the kth triplet as `(Ti[k], Tj[k], Tx[k])`. Thus, the matrix A of Section 12.4 may be encoded as

```
int N = 20, k = 0;    // N = 20 is an overestimate; N = 12 will do
int *Ti, *Tj;
double *Tx;
make_vector(Ti, N);
make_vector(Tj, N);
make_vector(Tx, N);
Ti[k] = 0; Tj[k] = 0; Tx[k] = 2.0; k++; // the (0,0) entry is 2.0
Ti[k] = 1; Tj[k] = 0; Tx[k] = 3.0; k++; // the (1,0) entry is 3.0
Ti[k] = 0; Tj[k] = 1; Tx[k] = 3.0; k++; // the (0,1) entry is 3.0
...
```

As you see, we are defining the sparse matrix here without constructing the full matrix. In UMFPACK, a representation of a matrix through the `Ti`, `Tj`, `Tx` vectors is called the *triplet form* of that matrix. Let me point out that the order in which the entries are defined is immaterial. For instance, the $(0, 1)$ entry is 3.0 no matter where it appears in the sequence of assignments above.

This brings up an issue to ponder: Suppose that we enter

```
Ti[5] = 3; Tj[5] = 4; Tx[5] = 2.0;   // the (3,4) entry is 2.0
Ti[8] = 3; Tj[8] = 4; Tx[8] = 9.0;   // the (3,4) entry is 9.0
```

One line says that the matrix's $(3,4)$ entry is 2.0. The other line says that it is 9.0. So which is it? UMFPACK answers the question by *adding the entries!* Thus, the two lines shown above are interpreted to mean that the $(3,4)$ entry is 11.0. In general, if an entry is given multiple times, its values accumulate. This is exactly what is needed in finite element programs. In Chapter 25, for instance, the matrix A (called the *system's stiffness matrix*) is constructed piecemeal. The information about its entries becomes available a little at a time and accumulates, just as in the snippet above.

12.6.2 ▪ From the triplet form to CCS

The triplet form is a temporary (but very useful) means of storing a sparse matrix. It needs to be translated to CCS because that's what UMFPACK's functions call for. UMFPACK's `umfpack_di_triplet_to_col()` function translates from the triplet form to CCS. Thus, suppose that the triplet vectors `Ti`, `Tj`, `Tx` vectors (of length N each) represent an $n \times n$ sparse matrix. To obtains A's CCS representation, we do

```
make_vector(Ap, n + 1);
make_vector(Ai, N);
make_vector(Ax, N);
status = umfpack_di_triplet_to_col(n, n, N, Ti, Tj, Tx, Ap, Ai, Ax, NULL);
if (status ≠ UMFPACK_OK)
    error_and_exit(status, __FILE__, __LINE__);
```

The call to `umfpack_di_triplet_to_col()` reads the triplet vectors `Ti`, `Tj`, `Tx` and populates the CCS vectors `Ap`, `Ai`, `Ax` with the corresponding data. The triplet vectors are not needed beyond this point, so you may free their memory now.

Remark 12.2. In view of the accumulation effect of the triplet representation, it should be clear that the number of triplets specified in the `Ti`, `Tj`, `Tx` vectors is *not* the same as the number of nonzero entries, `nz`, of the matrix A, which are fewer in general. Had I known `nz`, I would have allocated vectors of lengths `nz`, rather than N, for `Ai` and `Ax` in the code fragment above. But generally we don't know `nz` ahead of the time, so I used N instead. That's an overestimate and a bit wasteful but perhaps not worth fretting over too much.

Remark 12.3. Continuing on the issue brought up in the previous remark, recall that `Ap[n]` equals the number of nonzero entries in a CCS representation. Therefore we can tell a matrix's true `nz` value by looking up the number that is stored in `Ap[n]`. Unfortunately that information arrives too late—we need to allocate the vectors `Ai` and `Ax` of length N *before* calling `umfpack_di_triplet_to_col()`. Only then we may look up the value of `nz` in `Ap[n]`.

12.6.3 ▪ *umfpack-demo3.c*

Copy *umfpack-demo2.c* to *umfpack-demo3.c*. Remove the declaration "**`double`** `**a;`" and all references to the matrix `a`. Remove the header file *sparse.h* because it's no longer needed. In `main()`, replace the definition of the matrix A by its triplet form. Apply `umfpack_di_triplet_to_col()` to convert it to CCS. The rest of the file remains the same. It solves the system $Ax = b$ and prints the solution, as before.

12.7 ▪ *Project* UMFPACK

Part 12.1. Implement, compile, and test the programs *umfpack-demo1.c*, *umfpack-demo2.c*, and *umfpack-demo3.c*, as discussed in this chapter.

Part 12.2. The matrix A on our programs has an entry of 3.0 in its first column and second row. Copy *umfpack-demo3.c* to *umfpack-demo4.c* and modify *umfpack-demo4.c* so that the entry is specified twice: once as 1.0, and once again as 2.0. Entries in the triplet form accumulate; therefore that change should not affect the result. See if your program confirms that.

Chapter 13
Haar wavelets

Prerequisites: Chapters 7, 8

13.1 ▪ A brief background

Among the disconcerting findings in the nascent field of functional analysis in the late 19th century was the discovery of continuous functions on a closed interval whose (trigonometric) Fourier series converged pointwise but not uniformly; see Zygmund [84, Theorem 1.13, p. 300]. The generalization of the classical trigonometric Fourier series to the geometric abstraction of the Hilbert space in the first decade of the 20th century provided great impetus to study that and related pathological behaviors in a new light. Haar [24, 25], in a dissertation written under Hilbert, constructed a complete orthonormal basis for the Hilbert space $L^2(0,1)$ with the property that the (generalized) Fourier series in that basis of any continuous function converged uniformly to that function. This showed that the failure of uniform convergence noted above is due to the choice of trigonometric functions for the basis, rather than a generic property of all Fourier series.

Haar's basis functions gained a new-found prominence in the 1980s, where it was noted that they formed a very special case of the more general *theory of wavelets* that was being developed then. Thus, they were dubbed *Haar wavelets* retroactively.

I will not deal with general wavelets in this book. The present exposition is driven by the ultimate goal of applications to image analysis in Chapter 15. Haar wavelets are suited particularly well to that purpose; therefore I will limit the discussion to them.

Much of this chapter's material comes from the introductory articles of Stollnitz, DeRose, and Salesin [66, 67] that later developed into the book [68]. I have found Nievergelt's book [51] also quite informative and learned the "in place" wavelet transformation (see Section 13.6) from it. Strang's expository articles [69, 70] make for good readings as well. For algorithmic and programming issues dealing with general wavelets see [53].

13.2 ▪ The space $L^2(0,1)$

I will use the usual notation $L^2(0,1)$ for the linear space of square (Lebesgue) integrable real-valued functions on the interval $(0,1)$. The *inner product*$(\cdot,\cdot)$ and the *norm* $\|\cdot\|$ in $L^2(0,1)$ are defined by

$$(f,g)=\int_0^1 f(x)g(x)\,dx,\quad \|f\|^2=(f,f)=\int_0^1 |f(x)|^2\,dx,\quad f,g\in L^2(0,1).$$

It can be shown (see, e.g., [48] for a very readable account) that the norm satisfies the *triangle inequality*:

$$||f-g|| \le ||f-h|| + ||h-g|| \quad \text{for all } f, g, h \in L^2(0,1);$$

therefore $||f-g||$ is a *metric*, that is, a meaningful measure of "distance" between the functions f and g.

The inner product generalizes the idea of the dot product in $\boldsymbol{R}^n$. Thus, the functions f and g are said to be *orthogonal* in $L^2(0,1)$ if $(f,g)=0$. An *orthogonal set* $\mathscr{F} \subset L^2(0,1)$ is a collection of functions such that every pair of functions in $\mathscr{F}$ is orthogonal. Furthermore, an orthogonal set $\mathscr{F}$ is said to be *orthonormal* if $||f||=1$ for all $f \in \mathscr{F}$. Since it is possible to construct orthonormal sets in $L^2(0,1)$ that have arbitrarily many elements, $L^2(0,1)$ is an *infinite-dimensional* space.

In analogy with the decomposition of vectors in $\boldsymbol{R}^n$ along an orthonormal basis, we say the orthonormal set $\mathscr{B} = \{f_i\}_{i=1}^{\infty} \subset L^2(0,1)$ is a *complete orthonormal basis* in $L^2(0,1)$ if any function $f \in L^2(0,1)$ may be expressed as

$$f = \sum_{i=1}^{\infty} (f, f_i) f_i = (f, f_1) f_1 + (f, f_2) f_2 + \cdots, \tag{13.1}$$

where the convergence of the infinite sum is understood in the sense of the norm $||\cdot||$. The infinite sum here is called the *generalized Fourier series* of the function f with respect to the basis $\mathscr{B}$. One of the outcomes of Haar's 1910 article [24] is the explicit construction of a basis $\mathscr{B}$ of $L^2(0,1)$ such that the generalized Fourier series of any continuous function converges not only in the sense of the norm $||\cdot||$ but also *uniformly* to that function. We will study the details of Haar's construction in the next section.

13.3 ▪ Haar's construction

Pick an integer $j \ge 0$, and then partition the interval $[0,1]$ into 2^j subintervals of equal lengths through the dividing points with coordinates $i/2^j$, $i = 0, 1, \ldots, 2^j$. Let V_j be the set of piecewise-constant functions on $[0,1]$ whose discontinuities, if any, fall on those dividing points. Clearly V_j is a linear subspace of $L^2(0,1)$. Toward the construction of an orthonormal basis $\mathscr{B}_j$ in V_j, define $\phi : \boldsymbol{R} \to \boldsymbol{R}$ as[36]

$$\phi(x) = \begin{cases} 1 & \text{if } 0 \le x \le 1, \\ 0 & \text{otherwise,} \end{cases} \tag{13.2}$$

and for any $i = 0, 1, \ldots, 2^j - 1$, set

$$\phi_i^j(x) = 2^{j/2} \phi(2^j x - i), \quad 0 \le x \le 1, \tag{13.3}$$

and then let $\mathscr{B}_j = \{\phi_i^j\}_{i=0}^{2^j-1}$. (The j in ϕ_i^j is merely a *superscript*, not an *exponent*.) To understand $\mathscr{B}_j$, let us study the set $\mathscr{B}_2 = \{\phi_0^2, \phi_1^2, \phi_2^2, \phi_3^2\}$ first. The graphs of $\mathscr{B}_2$'s four functions are shown in the top row in Figure 13.1. It should be clear that any function in V_2 may be expressed uniquely as a linear combination of these; therefore $\mathscr{B}_2$ is a basis for V_2. In particular, the dimension of V_2, which is defined as the number of basis

[36] The values of $\phi(x)$ at $x=0$ and $x=1$ are irrelevant in wavelet applications and will be ignored. Those *were* relevant to Haar, whose aim was to prove the uniform convergence of the generalized Fourier series.

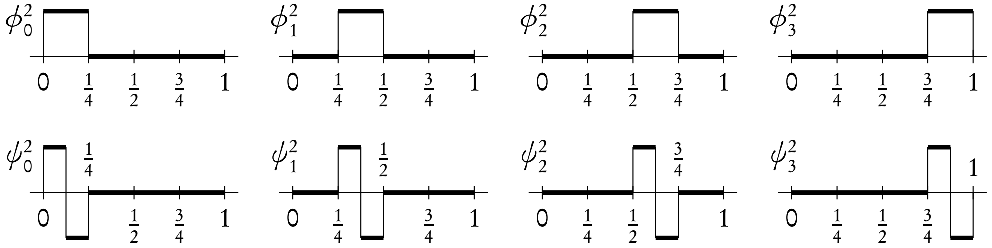


Figure 13.1: The Haar scaling functions $\{\phi_0^2, \phi_1^2, \phi_2^2, \phi_3^2\}$ form a basis for V_2. The Haar wavelets $\{\psi_0^2, \psi_1^2, \psi_2^2, \psi_3^2\}$ form a basis for W_2. Taken together, the eight functions form a basis for $V_3 = V_2 \oplus W_2$. Since $j = 2$, the height of the nonzero part of each of the functions shown above is $2^{j/2} = 2$. (These are not drawn to scale.) All function have unit L^2 norms.

elements, is 4. We write that as $\dim V_2 = 4$. Moreover, it should also be clear that the functions in $\mathscr{B}_2$ are orthogonal, that is, $(\phi_{i_1}^2, \phi_{i_2}^2) = \int_0^1 \phi_{i_1}^2(x)\phi_{i_2}^2(x)\,dx = 0$ if $i_1 \neq i_2$. This is because the regions where $\phi_{i_1}^2$ and $\phi_{i_2}^2$ are nonzero don't overlap.[37] The coefficient $2^{j/2}$ in (13.3) is chosen on purpose so that

$$\int_0^1 \left|\phi_i^j(x)\right|^2 dx = \left(2^{j/2}\right)^2 \times 2^{-j} = 1 \quad \text{for all } j.$$

Thus, $\mathscr{B}_2$ is an *orthonormal basis* in V_2.

The observations of the previous paragraph regarding V_2 generalize in a straightforward way to V_j for any $j \geq 0$. Thus, $\mathscr{B}_j$ is an orthonormal basis in V_j, and $\dim V_j = 2^j$.

The spaces V_j form a nested hierarchy of increasingly richer subspaces of $L^2(0,1)$:

$$V_0 \subset V_1 \subset V_2 \subset \cdots \subset L^2(0,1).$$

For each j, let W_j be the *orthogonal complement* of V_j in V_{j+1}. In other words, W_j is the set of all functions in V_{j+1} which are orthogonal to all functions in V_j. That's written as $V_{j+1} = V_j \oplus W_j$. Since $\dim V_{j+1} = 2^{j+1} = 2 \times 2^j$ and $\dim V_j = 2^j$, then $\dim W_j = 2^j$. We conclude that any function in the 2^{j+1}-dimensional space V_{j+1} may be decomposed uniquely into the sum of two functions in the 2^j-dimensional orthogonal subspaces V_j and W_j.

Toward constructing an orthonormal basis for W_j, let's define $\psi : \boldsymbol{R} \to \boldsymbol{R}$ as[38]

$$\psi(x) = \begin{cases} +1 & \text{if } 0 < x < \frac{1}{2}, \\ -1 & \text{if } \frac{1}{2} < x < 1, \\ 0 & \text{otherwise} \end{cases}$$

[37]Technical note: The closure of the set where a function is nonzero is called that function's *support*. We are saying that the supports of $\phi_{i_1}^2$ and $\phi_{i_2}^2$ don't overlap. This is not literally true because two such supports may have a point in common. But integration over a point produces zero; therefore the statement regarding the orthogonality of $\phi_{i_1}^2$ and $\phi_{i_2}^2$ still holds.

[38]As in the case of ϕ, the values of ψ at its points of discontinuity are irrelevant in wavelet applications. Haar gave them specific values because his aim was to study uniform convergence.

and, for any $i = 0, 1, \ldots, 2^j - 1$, set

$$\psi_i^j(x) = 2^{j/2}\psi(2^j x - i), \quad 0 \le x \le 1. \tag{13.4}$$

Let $\mathscr{B}_j' = \{\psi_i^j\}_{i=0}^{2^j-1}$. The graphs of the four functions $\mathscr{B}_2' = \{\psi_0^2, \psi_1^2, \psi_2^2, \psi_3^2\}$ are shown in the bottom row in Figure 13.1. It is clear that they form an orthogonal set, that is, $(\psi_{i_1}^2, \psi_{i_2}^2) = \int_0^1 \psi_{i_1}^2(x)\psi_{i_2}^2(x)\,dx = 0$ if $i_1 \ne i_2$. It is also clear that any of the functions in the bottom row is orthogonal to any of the function in the top row, that is, $(\psi_{i_1}^2, \phi_{i_2}^2) = \int_0^1 \psi_{i_1}^2(x)\psi_{i_2}^2(x)\,dx = 0$ for all i_1 and i_2. Moreover, the coefficient 2^j in (13.4) is chosen on purpose so that $\|\psi_i^j\| = 1$ for all i and j. Thus, $\mathscr{B}_2'$ is an *orthonormal set*, and span$\{\mathscr{B}_2'\}$ is orthogonal to V_2. Additionally, let us observe that every element of $\mathscr{B}_2'$ is in V_3. (Do you see that?)

The observations of the previous paragraph regarding the case $j = 2$ generalize in a straightforward way to any $j \ge 0$. Thus,

a. $\mathscr{B}_j'$ is an orthonormal set;

b. span$\{\mathscr{B}_j'\}$ is orthogonal to V_j; and

c. span$\{\mathscr{B}_j'\} \subset V_{j+1}$.

Recalling that W_j is the orthogonal complement of V_j in V_{j+1}, it follows that span$\{\mathscr{B}_j'\} \subset W_j$. But since span$\{\mathscr{B}_j'\}$ has 2^j elements, and since $\dim W_j = 2^j$, we conclude that span$\{\mathscr{B}_j'\} = W_j$, that is, $\mathscr{B}_j'$ is an orthonormal basis of W_j.

In the context of the modern theory of wavelets, the basis functions ϕ_i^j of V_j are called the *Haar scaling functions*, and the basis functions ψ_i^j of W_j are called the *Haar wavelets* on account of their wiggly graphs; see the graphs in the bottom row in Figure 13.1.

For future reference, let us note that the functions $2^{-j/2}\phi_i^j$ and $2^{-j/2}\psi_i^j$ both are of amplitude 1; therefore,

$$2^{-(j-1)/2}\phi_i^{j-1} = 2^{-j/2}\phi_{2i}^j + 2^{-j/2}\phi_{2i+1}^j,$$
$$2^{-(j-1)/2}\psi_i^{j-1} = 2^{-j/2}\phi_{2i}^j - 2^{-j/2}\phi_{2i+1}^j.$$

Solving for ϕ_i^{j-1} and ψ_i^{j-1} we get

$$\phi_i^{j-1} = \frac{1}{\sqrt{2}}(\phi_{2i}^j + \phi_{2i+1}^j), \quad i = 0, 1, \ldots, 2^{j-1} - 1, \tag{13.5a}$$

$$\psi_i^{j-1} = \frac{1}{\sqrt{2}}(\phi_{2i}^j - \phi_{2i+1}^j), \quad i = 0, 1, \ldots, 2^{j-1} - 1. \tag{13.5b}$$

These express the basis functions of V_{j-1} and W_{j-1} as linear combination of the basis functions of V_j. In effect, these are the concrete realizations of the decomposition $V_j = V_{j-1} \oplus W_{j-1}$. Figure 13.2 is an attempt to express this in a schematic diagram. Try to make a sense of it.

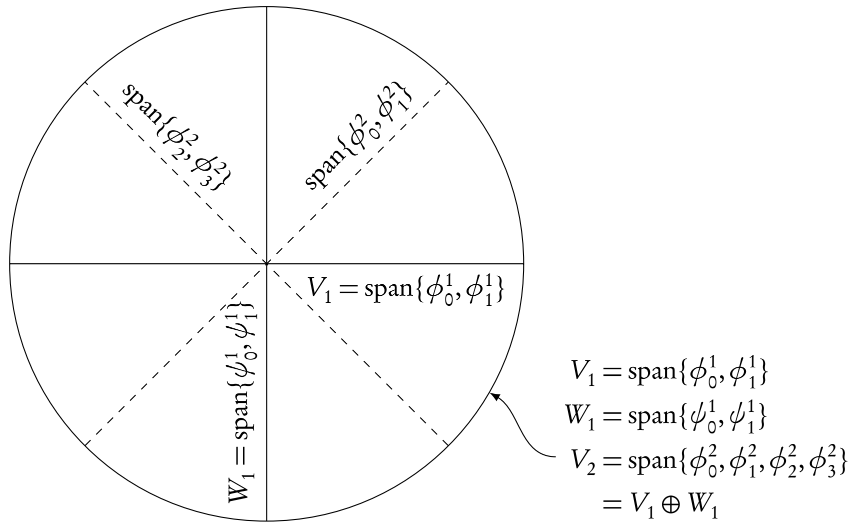


Figure 13.2: This schematic diagram is an attempt to illustrate the decomposition $V_2 = V_1 \oplus W_1$. The circle represents the four-dimensional space V_2. The horizontal and vertical lines represent the two-dimensional mutually orthogonal subspaces $V_1 = \text{span}\{\phi_0^1, \phi_1^1\}$ and $W_1 = \text{span}\{\psi_0^1, \psi_1^1\}$. The dashed lines represent V_2's natural basis $\{\phi_0^2, \phi_1^2, \phi_2^2, \phi_3^2\}$. Their rotated positions emphasize that they are not aligned with the spaces V_1 and W_1. Equations (13.5a) and (13.5b) give the rotation equations that relate the solid and dashed spaces.

13.4 ▪ The decomposition $V_j = V_{j-1} \oplus W_{j-1}$

Let us introduce the notation

$$\alpha_i^j = (f, \phi_i^j), \quad \beta_i^j = (f, \psi_i^j), \quad \text{where } f \in L^2(0,1). \tag{13.6}$$

In view of (13.5a) and (13.5b), we have

$$\alpha_i^{j-1} = (f, \phi_i^{j-1}) = \left(f, \frac{1}{\sqrt{2}}(\phi_{2i}^j + \phi_{2i+1}^j)\right) = \frac{1}{\sqrt{2}}(\alpha_{2i}^j + \alpha_{2i+1}^j),$$
$$\beta_i^{j-1} = (f, \psi_i^{j-1}) = \left(f, \frac{1}{\sqrt{2}}(\phi_{2i}^j - \phi_{2i+1}^j)\right) = \frac{1}{\sqrt{2}}(\alpha_{2i}^j - \alpha_{2i+1}^j).$$

This is a very significant result—it expresses the components of f along the bases $\mathscr{B}_{j-1}$ and $\mathscr{B}'_{j-1}$ in terms of those along the basis $\mathscr{B}_j$, and thus it offers yet another way of viewing the decomposition $V_j = V_{j-1} \oplus W_{j-1}$. Let us state it formally as a theorem.

Theorem 13.1. *For any function $f \in V_j$, the components α_i^j and β_i^j defined in* (13.6) *satisfy the recursive relations*

$$\alpha_i^{j-1} = \frac{1}{\sqrt{2}}(\alpha_{2i}^j + \alpha_{2i+1}^j), \quad i = 0, 1, \ldots, 2^{j-1} - 1, \tag{13.7a}$$

$$\beta_i^{j-1} = \frac{1}{\sqrt{2}}(\alpha_{2i}^j - \alpha_{2i+1}^j), \quad i = 0, 1, \ldots, 2^{j-1} - 1. \tag{13.7b}$$

Equations (13.7a) and (13.7b) may be written in matrix form as

$$\begin{pmatrix} \alpha_i^{j-1} \\ \beta_i^{j-1} \end{pmatrix} = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix} \begin{pmatrix} \alpha_{2i}^{j} \\ \alpha_{2i+1}^{j} \end{pmatrix}. \tag{13.8}$$

The 2×2 coefficient matrix is an *orthogonal matrix*;[39] therefore the equations may be inverted easily:

$$\begin{pmatrix} \alpha_{2i}^{j} \\ \alpha_{2i+1}^{j} \end{pmatrix} = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix} \begin{pmatrix} \alpha_i^{j-1} \\ \beta_i^{j-1} \end{pmatrix}. \tag{13.9}$$

This is also a very significant result—it expresses the components of $f \in V_j$ in the basis $\mathscr{B}_j$ in terms of its components in the bases $\mathscr{B}_{j-1}$ and $\mathscr{B}'_{j-1}$ of V_{j-1} and W_{j-1}. Let us state it formally as a theorem.

Theorem 13.2. *For any function $f \in V_j$, the components α_i^j and β_i^j defined in* (13.6) *satisfy the recursive relations*

$$\alpha_{2i}^{j} = \frac{1}{\sqrt{2}}(\alpha_i^{j-1} + \beta_i^{j-1}), \quad i = 0, 1, \ldots, 2^{j-1} - 1, \tag{13.10a}$$

$$\alpha_{2i+1}^{j} = \frac{1}{\sqrt{2}}(\alpha_i^{j-1} - \beta_i^{j-1}), \quad i = 0, 1, \ldots, 2^{j-1} - 1. \tag{13.10b}$$

13.5 ▪ From functions to vectors

In view of the notation $\alpha_i^j = (f, \phi_i^j)$ introduced in (13.6) and the general orthogonal expansion (13.1), any function $f \in V_j$ may be written as

$$f = \sum_{i=0}^{2^j-1} \alpha_i^j \phi_i^j. \tag{13.11}$$

This associates with any function $f \in V_j$ a unique vector $\boldsymbol{x} = [\alpha_0^j, \alpha_1^j, \ldots, \alpha_{n-1}^j] \in \boldsymbol{R}^n$, where $n = 2^j = \dim V_j$, and conversely, to any vector in $\boldsymbol{R}^n$ there corresponds a unique function in V_j. The association preserves the algebraic structure of spaces; that is, if f and g in V_j are associated with vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ in $\boldsymbol{R}^n$, then the function $af + bg$ corresponds to the vector $a\boldsymbol{x} + b\boldsymbol{y}$ for any $a, b \in \boldsymbol{R}$. Additionally, if we equip $\boldsymbol{R}^n$ with the Euclidean norm $\|\boldsymbol{x}\| = \left(\sum_{i=0}^{n-1} |\alpha_i^j|^2\right)^{1/2}$, then the association also preserves the metric structure of the spaces because then $(f, g) = \boldsymbol{x} \cdot \boldsymbol{y}$, and $\|f\| = \|\boldsymbol{x}\|$. Collectively, these properties are expressed by saying that the association between $f \in V_j$ and $\boldsymbol{x} \in \boldsymbol{R}^n$ is an *isomorphism*. In that sense, the spaces V_j and $\boldsymbol{R}^n$ (with $n = 2^j$) are indistinguishable as far as their algebraic and metric properties are concerned. In effect, any assertion regarding V_j maps to a corresponding assertion in $\boldsymbol{R}^n$, and vice versa.

At this point we part ways from Haar's original work. His focus was on the behavior of the function spaces V_j in the limit as $j \to \infty$. In the context of wavelets the focus is on the properties of V_j for a fixed j. Moreover, in view of the isomorphism between V_j

[39] A square matrix A is *orthogonal* if its product with its transpose is the identity matrix, that is, $A^T A = I$. It follows that an orthogonal matrix is invertible, and the inverse equals the transpose: $A^{-1} = A^T$.

and $\boldsymbol{R}^n$, our study will shift gradually from the function spaces V_j to the corresponding vector spaces $\boldsymbol{R}^n$. That shift is very evident in the following discussion.

Let us illustrate Theorems 13.1 and 13.2 in the case $j = 3$. Since $V_3 = V_2 \oplus W_2$, then an $f \in V_3$ may be expressed in two ways: (a) as a linear combination of the basis functions $\mathscr{B}_3$, or (b) as the sum of a linear combinations of the bases $\mathscr{B}_2$ and $\mathscr{B}_2'$:

$$f = \alpha_0^3\phi_0^3 + \alpha_1^3\phi_1^3 + \alpha_2^3\phi_2^3 + \alpha_3^3\phi_3^3 + \alpha_4^3\phi_4^3 + \alpha_5^3\phi_5^3 + \alpha_6^3\phi_6^3 + \alpha_7^3\phi_7^3,$$
$$f = \left(\alpha_0^2\phi_0^2 + \alpha_1^2\phi_1^2 + \alpha_2^2\phi_2^2 + \alpha_3^2\phi_3^2\right) + \left(\beta_0^2\psi_0^2 + \beta_1^2\psi_1^2 + \beta_2^2\psi_2^2 + \beta_3^2\psi_3^2\right).$$

The first representation is associated with the vector

$$\begin{bmatrix} \alpha_0^3 & \alpha_1^3 & \alpha_2^3 & \alpha_3^3 & \alpha_4^3 & \alpha_5^3 & \alpha_6^3 & \alpha_7^3 \end{bmatrix} \tag{13.12}$$

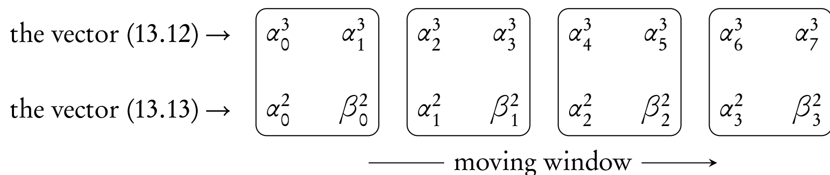
in $\boldsymbol{R}^8$. The second representation is associated with the pair of vectors $[\alpha_0^2, \alpha_1^2, \alpha_2^2, \alpha_3^2]$ and $[\beta_0^2, \beta_1^2, \beta_2^2, \beta_3^2]$ in $\boldsymbol{R}^4$. Let us merge the latter two vectors into a single vector in $\boldsymbol{R}^8$ by alternating their components, as in

$$\begin{bmatrix} \alpha_0^2 & \beta_0^2 & \alpha_1^2 & \beta_1^2 & \alpha_2^2 & \beta_2^2 & \alpha_3^2 & \beta_3^2 \end{bmatrix}. \tag{13.13}$$

Theorems 13.1 and 13.2 establish the transformation rule between the vectors (13.12) and (13.13). Specifically, according to Theorems 13.1,

$$\begin{pmatrix} \alpha_0^2 \\ \beta_0^2 \\ \alpha_1^2 \\ \beta_1^2 \\ \alpha_2^2 \\ \beta_2^2 \\ \alpha_3^2 \\ \beta_3^2 \end{pmatrix} = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & & & & & & \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & & & & & & \\ & & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & & & & \\ & & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & & & & \\ & & & & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & & \\ & & & & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & & \\ & & & & & & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ & & & & & & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix} \begin{pmatrix} \alpha_0^3 \\ \alpha_1^3 \\ \alpha_2^3 \\ \alpha_3^3 \\ \alpha_4^3 \\ \alpha_5^3 \\ \alpha_6^3 \\ \alpha_7^3 \end{pmatrix}.$$

Only the nonzero entries of the coefficient matrix are shown. The matrix's block-diagonal structure indicates that the transformation is *local* in the sense that the first pair of components in (13.13) is determined by the first pair in (13.12). Similarly, the second pair of components in (13.13) is determined by the second pair in (13.12), and so on. Thus, transforming the vector (13.12) to (13.13) may be performed by a "moving window" of width 2 as follows:



Here, within each window the entries in the top corners determine the entries in the bottom corners (Theorem 13.1), and conversely, the entries in the bottom corners determine the entries in the top corners (Theorem 13.2).

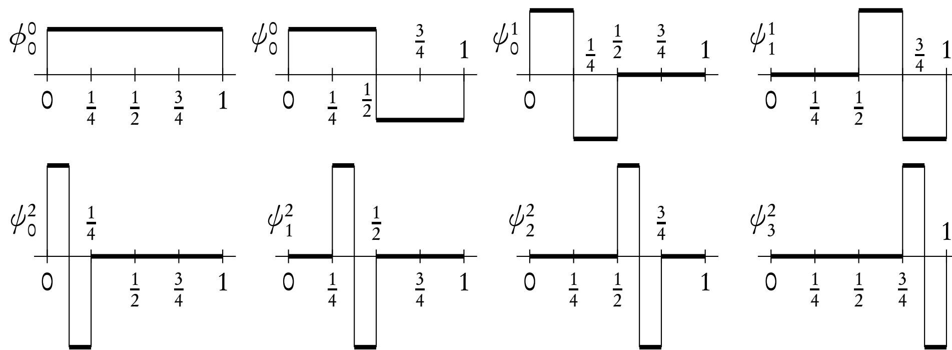


Figure 13.3: The wavelet basis $\mathscr{W}_3 = \{\phi_0^0, \psi_0^0, \psi_0^1, \psi_0^1, \psi_0^2, \psi_1^2, \psi_2^2, \psi_3^2\}$ of the eight-dimensional space V_3 shown to scale. The amplitudes of the first two are $2^{j/2} = 2^{0/2} = 1$. The amplitudes of the next two are $2^{j/2} = 2^{1/2} = \sqrt{2}$. The amplitudes of the last four are $2^{j/2} = 2^{2/2} = 2$. All function have unit L^2 norms.

13.6 ▪ The Haar wavelet transform

We have seen that $V_j = V_{j-1} \oplus W_{j-1}$. By the same token, $V_{j-1} = V_{j-2} \oplus W_{j-2}$. Substituting the V_{j-1} from the second equation into the first, we get $V_j = V_{j-2} \oplus W_{j-2} \oplus W_{j-1}$. By applying this idea repeatedly, we reduce V's index on the right-hand side to zero and arrive at

$$V_j = V_0 \oplus W_0 \oplus W_1 \oplus \cdots \oplus W_{j-1}. \tag{13.14}$$

This expression is called *the wavelet decomposition of the space* V_j. The union of the bases of the individual parts, $\mathscr{W}_j = \mathscr{B}_0 \cup \mathscr{B}_0' \cup \mathscr{B}_1' \cup \cdots \cup \mathscr{B}_{j-1}'$, is an orthonormal basis in V_j and is called the *Haar wavelet basis* of the space V_j. When a function $f \in V_j$ is expressed as a linear combination of the Haar wavelet basis, the coefficients of that linear combination are called the function's *Haar wavelet coefficients*. For instance, if $f \in V_3 = V_0 \oplus W_0 \oplus W_1 \oplus W_2$, then

$$f = \underbrace{\alpha_0^0\, \phi_0^0}_{\in V_0} + \underbrace{\beta_0^0\, \psi_0^0}_{\in W_0} + \underbrace{\beta_0^1\, \psi_0^1 + \beta_1^1\, \psi_1^1}_{\in W_1} + \underbrace{\beta_0^2\, \psi_0^2 + \beta_1^2\, \psi_1^2 + \beta_2^2\, \psi_2^2 + \beta_3^2\, \psi_3^2}_{\in W_2}$$

and the wavelet coefficients are $[\alpha_0^0, \beta_0^0, \beta_0^1, \beta_1^1, \beta_0^2, \beta_1^2, \beta_2^2, \beta_3^2]$. Figure 13.3 shows the graphs of the Haar wavelet basis functions in $\mathscr{W}_3$. *The purpose of the rest of this section is to obtain a simple algorithm for computing the Haar wavelet coefficients of functions* $f \in V_j$ *for any* j.

Remark 13.1. The space V_0 consists of constant functions on the interval $[0,1]$. Each wavelet in W_i is a function of zero average. Therefore in the decomposition (13.14) the component in V_0 picks up f's average over the interval $[0,1]$. The wavelet components add progressively finer details to that average and ultimately reconstruct f.

13.6.1 ▪ Computing the wavelet coefficients

Theorem 13.1 tells us how to decompose V_j into $V_{j-1} \oplus W_{j-1}$. The full wavelet decomposition in (13.14) is a matter of repeated application of that theorem. I will explain the

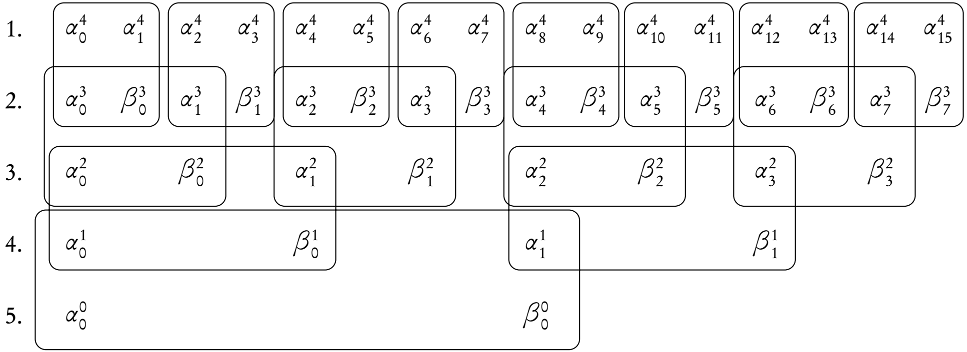


Figure 13.4: Rows 1 through 5 show the various stages of an *in place* Haar transform of an $f \in V_4$. Row 1 is the vector of the 16 coefficients of the expansion of f in the basis $\mathscr{B}_4$. To reduce clutter, only the vector's changed elements are shown in each row. Blanks indicate values that are to be carried over from a previous row.

procedure in the case of $j = 4$, that is,

$$V_4 = V_0 \oplus W_0 \oplus W_1 \oplus W_2 \oplus W_3.$$

The generalization to arbitrary j should should be evident.

Figure 13.4 illustrates the algorithm. It begins with a function $f = \sum_{i=0}^{15} \alpha_i^4 \phi_i^4 \in V^4$. The associated vector in $\boldsymbol{R}^{16}$, that is,

$$\begin{bmatrix} \alpha_0^4 & \alpha_1^4 & \alpha_2^4 & \alpha_3^4 & \alpha_4^4 & \alpha_5^4 & \alpha_6^4 & \alpha_7^4 & \alpha_8^4 & \alpha_9^4 & \alpha_{10}^4 & \alpha_{11}^4 & \alpha_{12}^4 & \alpha_{13}^4 & \alpha_{14}^4 & \alpha_{15}^4 \end{bmatrix}, \tag{13.15}$$

is entered in row 1 in Figure 13.4. The algorithm that I will describe changes that vector into its Haar wavelet transform *in place*; that is, the vector is overwritten by the transform. No extra storage is required. Rows 1 through 5 show the successive stages of the conversion of that vector. To eliminate clutter, each row shows only the parts of the vector that are changed relative to the previous row. The unchanged parts are left blank. For instance, there are only two changed entries in row 5. The remaining 14 entries (which are not shown) are as they were in the previous rows.

The procedure begins with the "moving window" idea, described in the previous section, applied to the vector of row 1 to produce the vector of row 2. This amounts to the decomposition $V_4 = V_3 \oplus W_3$. The α_i^3 and β_i^3 in row 2 are the components of f in V_3 and W_3. The α_i^3 entries await further work since V_3 is to be decomposed into $V_2 \oplus W_2$. The β_i^3 entries are finalized at this stage. In the subsequent rows their positions are left blank to indicate that their previous values carry over.

The moving window that takes us from row 2 to row 3 moves in steps of 4 because it is interested in the α_i^3 coefficients only—it skips over the β_i^3's. Similarly, the moving window that takes us from row 3 to row 4 moves in steps of 8. Finally the moving window that takes us from row 4 to row 5 moves in steps of 16, so it doesn't move at all. It finalizes the coefficients α_0^0 and β_0^0 and ends the process. The overall outcome of the process is the vector

$$\begin{bmatrix} \alpha_0^0 & \beta_0^3 & \beta_0^2 & \beta_1^3 & \beta_0^1 & \beta_2^3 & \beta_1^2 & \beta_3^3 & \beta_0^0 & \beta_4^3 & \beta_2^2 & \beta_5^3 & \beta_1^1 & \beta_6^3 & \beta_3^2 & \beta_7^3 \end{bmatrix}. \tag{13.16}$$

The transformation of the vector (13.15) into (13.16) is called the *Haar wavelet transform* on V_4. The transformation may be reversed since every step of the procedure is reversible. Going from (13.16) to (13.15) is called the *inverse Haar wavelet transform*.

13.6.2 ▪ An extra twist

Throughout this section I have worked with functions $f \in V_j$ expressed in the basis $\mathscr{B}_j$ of V_j, as in (13.11). In practice, however, a function $f \in V_j$ is *not* given in terms of the $\mathscr{B}_j$ basis. It is more likely given as a list of values that it takes on the 2^j subintervals of the interval $[0,1]$, as in

$$f(x) = c_i \qquad 2^{-j} i < x < 2^{-j}(i+1), \qquad i = 0, 1, \ldots, 2^j - 1.$$

So to get started with the transformation algorithms described earlier, the first step would be to calculate the coefficients α_i^j from the values c_i. This is quite easy, actually. Since the amplitude of each ϕ_i^j is $2^{j/2}$, then c_j and α_i^j are related through $c_j = 2^{j/2}\alpha_i^j$, or equivalently,

$$\alpha_i^j = \frac{1}{\sqrt{\dim V_j}} c_j, \tag{13.17}$$

because $\dim V_j = 2^j$. In words, the coefficients $\alpha_0^j, \alpha_1^j, \ldots$ of the expansion of f in the $\mathscr{B}_j$ basis of V_j are obtained simply by dividing the list of the f's values by $\sqrt{\dim V_j}$.

13.7 ▪ Functions of two variables

The study of the previous sections led to the construction of the wavelet basis $\mathscr{W}_j$ of piecewise constant functions defined on the interval $[0,1]$ partitioned into 2^j equal subintervals. Figure 13.3 shows the $2^3 = 8$ members of $\mathscr{W}_3$.

The two-dimensional generalization partitions the square $[0,1]^2$ through a uniform $2^{j_1} \times 2^{j_2}$ rectangular grid, where j_1 and j_2 are nonnegative integers, and considers functions $f : (x,y) \in [0,1]^2 \to \boldsymbol{R}$ which are constant on each cell of the grid.

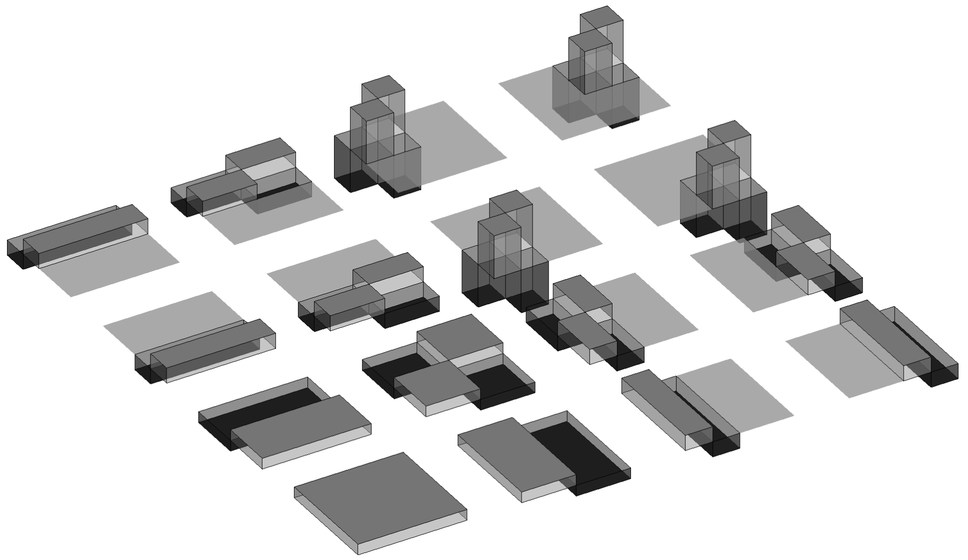
A basis for the resulting function space is given by the functions

$$\mathscr{W}_{j_1,j_2} = \Big\{ h(x,y) = f(x)g(y) \,:\, f \in \mathscr{W}_{j_1},\ g \in \mathscr{W}_{j_2} \Big\}.$$

Figure 13.5 shows the 16 elements of the basis $\mathscr{W}_{2,2}$ from two different perspectives.

The theory of wavelets in two dimensions—indeed, in n dimensions—proceeds along lines similar to what we have seen in one dimension. I will not go through the details here but will refer you to the very readable accounts in [68] and [51]. After all is said and done, it turns out that computing the Haar wavelet transform of functions in two dimensions reduces to the following. Represent the function $f(x,y)$ as a $2^{j_1} \times 2^{j_2}$ matrix A which holds the (constant) values of f on the cells of the grid described above. Then do the following:

- Apply the one-dimensional Haar wavelet transform to every row of the matrix A. Let T be the resulting matrix.
- Apply the one-dimensional Haar wavelet transform to every column of the matrix T. The result is the Haar transform of A.



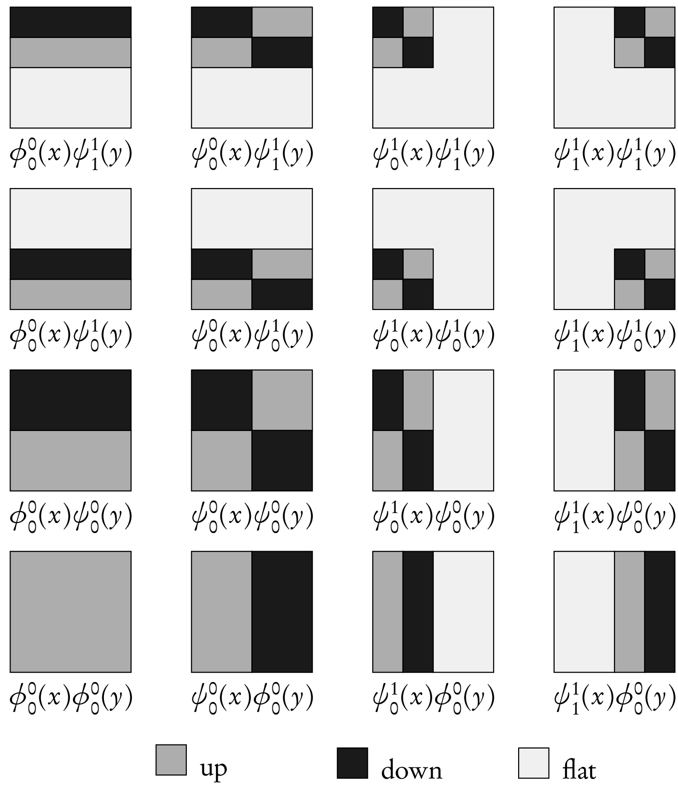


Figure 13.5: Two different views of the 16 basis functions of $\mathscr{W}_{2,2}$. At the top is a perspective view of their graphs. The relative heights of the blocks are drawn to scale. At the bottom is the "bird's eye view" of those graphs.

Listing 13.1: The header file *wavelet.h*.

```
1 #ifndef H_WAVELET_H
2 #define H_WAVELET_H
3 #define WT_FWD  0
4 #define WT_REV  1
5 void haar_transform_vector(double *v, int n, int dir);
6 void haar_transform_matrix(double **a, int m, int n, int dir);
7 #endif /* H_WAVELET_H */
```

The order of application of the two steps is immaterial. That is, you may transform all the columns first and then transform all the rows. Either way you will arrive at the same answer. For the inverse transform, just reverse the process.

13.8 ▪ An overview of the *wavelet* module

Our *wavelet* module provides functions for applying the Haar wavelet transform, and its inverse, to vectors and matrices. The files *wavelet.c* and *wavelet.h* provide the module's implementation and the interface. The module itself is self-contained in the sense that it has no need for anything else that we have developed in this book. The testing of the module, however, calls for allocating and freeing vectors and matrices. We do those with the help of the `xmalloc()` function developed in Chapter 7 and the header file *array.h* developed in Chapter 8. Therefore, following the suggestions of Chapters 2 and 6, your project's directory will look like this:

```
$ cd wavelets
$ ls -F
Makefile   wavelet-test.c  wavelet.h   xmalloc.h@
array.h@   wavelet.c       xmalloc.c@
```

The file *wavelet-test.c* provides a test/demonstration of our wavelet routines. You will see what it is about, and the output of a test run, in this chapter's *Projects* section. I will describe the contents of the files *wavelet.h* and *wavelet.c* next.

13.9 ▪ The file *wavelet.h*

Listing 13.1 shows the header file *wavelet.h* in its entirety. It declares two functions:

- `haar_transform_vector()` receives a pointer to a vector `v` of length `n` and applies the forward or reverse Haar transform to it, depending on the setting of the third argument, `dir` (short for *direction*), which can be one of `WT_FWD` or `WT_REV`.
- `haar_transform_matrix()` receives a pointer to an `m`×`n` matrix and applies the forward or reverse Haar transform to it, depending on the setting of the third argument, which can be one of `WT_FWD` or `WT_REV`.

Any other value passed for the third argument is an error; these functions will print an error message to the `stderr` and call `exit()` to terminate the program in that case. Furthermore, the following hold:

- It is *assumed* that `m` and `n` are powers of 2; we don't check for that!
- The functions do their work in situ; that is, they *overwrite* the given vector or matrix with the requested transform. If you are going to need the original object afterward, save a copy before calling these functions.

Listing 13.2: An outline of the file *wavelet.c*.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4  #include "wavelet.h"
5  #define SQRT1_2         sqrt(1.0/2)
6   ▶static void haar_transform_vector_forward(double *v, int n) ...
7   ▶static void haar_transform_vector_reverse(double *v, int n) ...
8   ▶static void haar_transform_matrix_forward(double **a, int m, int n) ...
9   ▶static void haar_transform_matrix_reverse(double **a, int m, int n) ...
10  ▶void haar_transform_vector(double *v, int n, int dir) ...
11  ▶void haar_transform_matrix(double **a, int m, int n, int dir) ...
```

Remark 13.2. The values `0` and 1 of the preprocessor symbol `WT_FWD` or `WT_REV` defined in *wavelet.h* are not significant; any two distinct integers will do.

13.10 ▪ The file *wavelet.c*

Listing 13.2 shows an outline of the file *wavelet.c*. The *math.h* header file is included to provide a declaration of the C library function `sqrt()`, which is used in several places in the code. The special case of $1/\sqrt{2}$ is handled separately by assigning it to a macro named `SQRT1_2`. An optimizing compiler will recognize that `SQRT1_2` evaluates to a constant and therefore will replace all occurrences of `SQRT1_2` in the program with the corresponding numerical value *at the time when the program is compiled*. `Gcc` works that way. If your compiler doesn't, then it may evaluate $1/\sqrt{2}$ thousands of times during the program's execution and thus degrade its performance. If you face that problem, change the definition of `SQRT1_2` to

```
#define SQRT1_2     0.70710678118654752440      /* sqrt(1.0/2) */
```

The implementation's front end is provided by the two functions defined on lines 10 and 11, which we met in the previous section. These enlist the help of the four functions defined on lines 6 through 9, which do the real work, that is, apply the forward or reverse Haar wavelet transform to vectors and matrices. Their names clearly indicate which does what. All four functions are specified **`static`**, and thus they are not accessible from outside *wavelet.c*.

13.10.1 ▪ The function `haar_transform_vector()`

The implementation of `haar_transform_vector()` that appears on line 10 of Listing 13.2 is quite straightforward; it is shown in its entirety in Listing 13.3. You will write the equally simple `haar_transform_matrix()`, which appears on line 11.

13.10.2 ▪ The function `haar_transform_vector_forward()`

The function `haar_transform_vector_forward()` that appears on line 6 of Listing 13.2 receives a pointer `v` to a vector of length `n` and applies the algorithm of Section 13.6 to convert that vector to the corresponding Haar wavelet transform. The function is *destructive* in the sense that it overwrites the original vector, but it has the

Listing 13.3: The function `haar_transform_vector()` calls one of two auxiliary functions to do the real transformation work. It expects to be called with the third argument set to one of `WT_FWD` or `WT_REV`; otherwise it prints an error message and calls `exit()`.

```
1  void haar_transform_vector(double *v, int n, int dir)
2  {
3      if (dir == WT_FWD)
4          haar_transform_vector_forward(v, n);
5      else if (dir == WT_REV)
6          haar_transform_vector_reverse(v, n);
7      else {    // shouldn't be here!
8          fprintf(stderr, "*** error in haar_transform_vector(): "
9                  "the third argument should be one of "
10                 "WT_FWD or WT_REV\n");
11         exit(EXIT_FAILURE);
12     }
13 }
```

Listing 13.4: The function `haar_transform_vector_forward()`.

```
1  static void haar_transform_vector_forward(double *v, int n)
2  {
3      double h = sqrt(n);
4      int i, d;
5      for (i = 0; i < n; i++)
6          v[i] /= h;
7      for (d = 1; d < n; d *= 2)
8          for (i = 0; i < n; i += 2*d) {
9              double x = SQRT1_2 * (v[i] + v[i+d]);
10             double y = SQRT1_2 * (v[i] - v[i+d]);
11             v[i] = x;
12             v[i+d] = y;
13         }
14 }
```

advantage of not requiring extra storage to do its work. The length of the vector, `n`, is assumed to be a power of 2; the function does not check for that!

Before getting into the details of the function's implementation, let us recall the discussion in Section 13.5 regarding the isomorphism between (piecewise constant) functions in V_j and vectors in $\boldsymbol{R}^n$, where $n = 2^j$.

A function in $f \in V_j$ is identified by the values it take on the 2^j subintervals of the interval of $[0,1]$. Those values form a vector $v = [c_0, c_1, \ldots, c_{2^j-1}]$, which is what the function `haar_transform_vector_forward()` receives. The first order of things is to apply the formula (13.17) (page 102) to calculate the components α_i^j of f in V_j. The rest of the algorithm deals with the α_i^j coefficients.

The complete implementation of the function is shown in Listing 13.4. Note the application of the formula (13.17) at the very outset on line 6.

The transformation algorithm begins on line 7. The index d begins with 1 and doubles its value on each iteration. For instance, when $n = 16$, as in the example of Figure 13.4 on page 101, then d takes on the values 1, 2, 4, 8. These correspond, respectively, to

transforming lines 1 to 2, 2 to 3, 3 to 4, and 4 to 5 in that figure. The inner loop on line 8 moves the "moving windows" within each row in steps of $2d$. The width of the window is $d+1$.

To understand lines 9 and 10, let us focus on the specific case of $d = 4$ and $i = 0$. This corresponds to the first of the two moving windows that straddle lines 3 and 4 of Figure 13.4. (The other window on that row corresponds to the next value of i which is $2d$, that is, 8.) It's the job of that window to read the values of α_0^2 and α_1^2 (at its upper corners) and compute the values of α_0^1 and β_0^1 (at its lower corners) through the formulas (13.7a) and (13.7b) in Theorem 13.1. Note that α_0^2 is located at the 0th place (which is the ith place) in vector v, and α_1^2 is located at the 4th place (which is the $(i+d)$th place) in vector v. This, together with Theorem 13.1, should explain the calculations on lines 9 and 10. Subsequently, on lines 11 and 12, the calculated values are inserted in the ith and $(i+d)$th places in the vector v, completing that step of the moving window's operation.

Something to think about: Why are the calculations on lines 9 and 10 separated from the insertions on lines 11 and 12? What if we do away with the intermediate x and y and collapse those four lines into two:

```
v[i]   = SQRT1_2 * (v[i] + v[i+d]);
v[i+d] = SQRT1_2 * (v[i] - v[i+d]);
```

Is there anything wrong with that?

13.10.3 ▪ The function `haar_transform_vector_reverse()`

The function `haar_transform_vector_reverse()` that appears on line 7 of Listing 13.2 implements the inverse Haar wavelet transform—it receives a pointer v to a vector of length n and applies the algorithm of Section 13.6 in reversed steps. The function is *destructive* in the sense that it overwrites the original vector, but it has the advantage of not requiring extra storage to do its work. The length of the vector, n, is assumed to be a power of 2. The function does not check for it!

The successive applications of `haar_transform_vector_forward()` followed by `haar_transform_vector_reverse()` to a vector should leave that vector unchanged (other than inaccuracies due to floating point arithmetic).

I will leave it to you to implement `haar_transform_vector_reverse()`. It's a matter of reversing the order of computations of Listing 13.4. Let me point out that lines 9 and 10, which reflect Theorem 13.1, should be replaced, in principle, by those from Theorem 13.2. However, upon a closer scrutiny we see that the transformation matrices in (13.8) and (13.9) are identical! As a result, lines 9 through 12 of Listing 13.4 carry over without change to the new function.

13.10.4 ▪ The function `haar_transform_matrix_forward()`

Following the ideas introduced in Section 13.7, consider a partitioning of the square $[0,1]\times[0,1]$ into an $m \times n$ grid of identical rectangular cells, where $n = 2^{j_1}$ and $m = 2^{j_2}$ for some nonnegative integers j_1 and j_2. A piecewise constant function on that gird is identified by the $m \times n$ matrix of the values that it takes on the grid's cells. In Section 13.7 we learned that the Haar wavelet transform of such a function is obtained by applying the one-dimensional Haar wavelet transform to every row and then to every column of the matrix.

The function `haar_transform_matrix_forward()` that appears on line 8 of Listing 13.2 receives a pointer a to such an m×n matrix and overwrites the matrix with

the corresponding Haar wavelet transform. It expects m and n to be powers of 2; *it does not check for that!*

At first glance, the implementation of haar_transform_matrix_forward() appears to be trivial; it's a matter of passing the matrix's rows and columns to the one-dimensional haar_transform_vector(), as in

```
for (i = 0; i < m; i++)                               // transform rows
    haar_transform_vector(a[i], n, WT_FWD);
for (j = 0; j < n; j++)                               // transform columns
    haar_transform_vector(... column j (how?) ...);
```

We see that transforming the rows is trivial indeed since a[i] points to the vector that makes up the matrix's row i. Transforming the columns is not so straightforward because the matrix's columns cannot be accessed as vectors.

There are at least two ways around this difficulty. One way is to set aside an auxiliary vector of length m, copy the values of the matrix's column j into the vector, pass the vector to haar_transform_vector(), and then copy the values from the vector back into the matrix:

```
for (i = 0; i < m; i++)                               // transform rows
    haar_transform_vector(a[i], n, WT_FWD);
for (j = 0; j < n; j++) {                             // transform columns
    for (i = 0; i < m; i++)                           // step 1: copy column to tmp
        tmp[i] = a[i][j];
    haar_transform_vector(tmp, m, WT_FWD);            // step 2: transform tmp
    for (i = 0; i < m; i++)                           // step 3: copy tmp to column
        a[i][j] = tmp[i];
}
```

Another way of transforming columns is to forgo haar_transform_vector() altogether. Instead, duplicate the code from haar_transform_vector() and customize it for the specific purpose of transforming matrix columns:

```
for (i = 0; i < m; i++)                               // transform rows
    haar_transform_vector(a[i], n, WT_FWD);
h = sqrt(m);                                          // transform columns
for (j = 0; j < n; j++) {
    // customized version of haar_transform_vector()
    for (i = 0; i < m; i++)
        a[i][j] /= h;
    for (d = 1; d < m; d*=2)
        for (i = 0; i < m; i += 2*d) {
            double x = SQRT1_2 * (a[i][j] + a[i+d][j]);
            double y = SQRT1_2 * (a[i][j] - a[i+d][j]);
            a[i][j] = x;
            a[i+d][j] = y;
        }
}
```

I don't have a strong preference of one method over the other. The first method has a cleaner code, as it relies entirely on haar_transform_vector() to do its work. However, it requires allocating and freeing an auxiliary vector. Furthermore, the back-and-forth copying of the matrix's columns into the auxiliary vector is ugly and wasteful. The second method avoids those disadvantages; however, the wisdom of the code duplication is questionable and certainly ugly. I leave it to you to decide between the two

methods. Pick one, and write a `haar_transform_matrix_forward()` for your *wavelet.c*.

13.10.5 ▪ The function `haar_transform_matrix_reverse()`

The function `haar_transform_matrix_reverse()` that appears on line 9 of Listing 13.2 implements the two-dimensional inverse Haar wavelet transform—it receives a pointer `a` to an `m×n` matrix and applies the one-dimensional inverse wavelet transform to every column and then to every row. The result overwrites the original matrix. It expects `m` and `n` to be powers of 2; *it does not check for that!* There is not much else I can add regarding this function; it's just like the forward transform.

13.11 ▪ *Project Wavelets*

Your wavelets module should be ready to test now. We are going to write a test/demo program, let's call it *wavelet-test.c*, to exercise that module and verify that it works correctly. Listing 13.5 gives an outline of *wavelet-test.c*. The function `test_vector()` that begins on line 4 makes a vector of length n (assuming n is a power of 2) and sets its entries to $v_i = \frac{1}{1+i}$, $i = 0, 1, \ldots$. It prints the vector to `stdout`, then it applies the Haar wavelet transform to it and prints the result, and then it applies the inverse Haar wavelet transform to it and prints the result again. If the program works correctly, then the first and third vectors will be the same. Vectors and matrices are printed by calling `print_vector()` and `print_matrix()` defined in Chapter 8's *array.h*.

The function `test_matrix()` on line 28 performs a similar test on matrices. Specifically, `test_matrix()` makes an $m \times n$ matrix (assuming that m and n are powers of 2), sets its entries to $a_{ij} = \frac{1}{1+i+j}$, and then prints the matrix, its wavelet transform, and its inverse wavelet transform. If the program works correctly, then the first and third matrices will be the same. Listing 13.6 shows the output of my version of the program.

Listing 13.5: An outline of *wavelet-test.c*.

```
1  #include <stdio.h>
2  #include "array.h"
3  #include "wavelet.h"
4  static void test_vector(int n)
5  {
6      double *v;
7
8      make_vector(v, n);
9      for (int i = 0; i < n; i++)
10         v[i] = 1.0/(1+i);
11
12     printf("original vector:\n");
13     print_vector("%8.4f ", v, n);
14     putchar('\n');
15
16     haar_transform_vector(v, n, WT_FWD);
17     printf("transformed vector:\n");
18     print_vector("%8.4f ", v, n);
19     putchar('\n');
20
21     haar_transform_vector(v, n, WT_REV);
22     printf("reconstructed vector:\n");
23     print_vector("%8.4f ", v, n);
24     putchar('\n');
25
26     free_vector(v);
27 }
28  ▶ static void test_matrix(int m, int n)
29 {
30     ... a[i][j] = 1.0/(1+i+j) ...
31 }
32 int main(void)
33 {
34     test_vector(8);        // test an 8-vector
35     test_matrix(4,8);      // test a 4×8 matrix
36     return 0;
37 }
```

Listing 13.6: The output of *wavelet-test*.

```
original vector:
  1.0000  0.5000  0.3333  0.2500  0.2000  0.1667  0.1429  0.1250

transformed vector:
  0.3397  0.1250  0.1620  0.0208  0.1811  0.0083  0.0175  0.0045

reconstructed vector:
  1.0000  0.5000  0.3333  0.2500  0.2000  0.1667  0.1429  0.1250

original matrix:
  1.0000  0.5000  0.3333  0.2500  0.2000  0.1667  0.1429  0.1250
  0.5000  0.3333  0.2500  0.2000  0.1667  0.1429  0.1250  0.1111
  0.3333  0.2500  0.2000  0.1667  0.1429  0.1250  0.1111  0.1000
  0.2500  0.2000  0.1667  0.1429  0.1250  0.1111  0.1000  0.0909

transformed matrix:
  0.2238  0.0500  0.0732  0.0119  0.0934  0.0056  0.0121  0.0032
  0.0393  0.0295  0.0333  0.0029  0.0314  0.0008  0.0016  0.0004
  0.0604  0.0333  0.0417  0.0048  0.0432  0.0016  0.0031  0.0007
  0.0107  0.0029  0.0048  0.0008  0.0061  0.0004  0.0007  0.0002

reconstructed matrix:
  1.0000  0.5000  0.3333  0.2500  0.2000  0.1667  0.1429  0.1250
  0.5000  0.3333  0.2500  0.2000  0.1667  0.1429  0.1250  0.1111
  0.3333  0.2500  0.2000  0.1667  0.1429  0.1250  0.1111  0.1000
  0.2500  0.2000  0.1667  0.1429  0.1250  0.1111  0.1000  0.0909
```

Chapter 14
Image I/O

Prerequisites: *Chapters 7, 8*

14.1 ▪ Digital images and image file formats

A digital image is made up of tiny colored dots called *pixels*.[40] Colors vary from one pixel to the next, but within each pixel the color is fixed. Almost always one thinks of a digital image as an arrangement of pixels on a *rectangular* grid—and that's what we do in this chapter—but other possibilities exist. An arrangement of pixels on a *hexagonal* grid may have some advantages (see [47] for a survey), but as far as I know, no hardware exists to display such images.

Digital images seem to be everywhere you look nowadays. A typical computer screen has between 1000 to 2000 pixels horizontally, and 700 to 1500 vertically. Some high-resolution monitors go substantially beyond that. A "five megapixel" digital camera produces 2560 × 1920 images:

$$2560 \times 1920 = 4{,}915{,}200 \approx 5 \text{ million pixels} = 5 \text{ megapixels}.$$

The resolution of a laser printer is measured in "dots per inch" (dpi). Ordinary ones tend to do 300 dpi or 600 dpi, but 1200 dpi and 2400 dpi ones are also available.

A rectangular image maps quite naturally to a (mathematical) matrix, but there are a variety of ways of mapping that matrix into a computer file (or a digital camera's memory chip). That results in the variety of image formats—BMP, GIF, PNG, JPG, TIFF, etc.—that you may encounter daily. I cannot afford to talk about all possible formats. Instead, I will focus on the *Netpbm* family of image formats, which were introduced by Jeffrey Poskanzer in the mid-1980s and developed into the current *Netpbm*[41] suite of image manipulation utilities and libraries about which we will learn in this chapter.

Netpbm recognizes three distinct image types:

- In a *Portable Bitmap* (PBM) image, pixels are either black or white (or any other pair of colors).
- In a *Portable Graymap* (PGM) image, pixel colors are shades of gray depending on the pixel's *gray value*, which is an integer g in the range $0 \le g \le M$ for some M

[40] The word "pixel" is a combination of "pix" (slang for "picture") and "element"; thus "picture element".

[41] *Netpbm* is a free and *open source* software. It's quite portable and works on all of the currently popular operating systems.

($M = 255$ being quite common). A pixel of gray value $g = 0$ is absolutely black, and a pixel of gray value $g = M$ is maximally white. The in-between gray values correspond to varying shades of gray.

- In a *Portable Pixmap* (PPM) image, pixels range over a wide spectrum of colors. A pixel's color is specified as an integer triplet (r, g, b) of the intensities of its red, green, and blue components, where $0 \leq r, g, b \leq M$ for some M ($M = 255$ being quite common).

Each of the three image formats has a "plain" and "raw" variant. They are closely related. The storage of the plain variant is entirely text based—it is trivial to inspect and edit such image data with an ordinary text editor—but it is quite wasteful in disk usage. The raw variant is stored in the binary format and is more frugal in disk usage. In the following sections I will describe the details of *Netpbm*'s PBM, PGM, and PPM image file formats.

The *Netpbm* library offers C functions to read and write images. It's this chapter's objective to learn how to use those library functions. You will need a properly installed *Netpbm* library[42] on your computer in order to compile this chapter's programs.

Remark 14.1. The formats of the PBM, PGM, and PPM image files are so simple that to appeal to a specialized library to read and write them is somewhat an overkill. Nevertheless, the *Netpbm* library is there, and interfacing with it provides a good learning experience. I must add that what we see in this chapter is a *minuscule* part of what that library offers. To learn more about the library's capabilities you will have to read its documentation, which unfortunately is rather disorganized and not quite reader-friendly.

Remark 14.2. You may use your favorite image viewer to display this chapter's images on your computer screen. Most viewers recognize the PBM, PGM, and PPM image formats. Under Linux I use the `feh` image viewer most of the time. Occasionally I use the `display` utility that comes with the *ImageMagick* utilities and the venerable `xv`, which, unfortunately, is no longer actively maintained.

Remark 14.3. I have provided a small collection of PBM, PGM, and PPM images in the book's website at `<www.siam.org/books/cs13>`, which you are invited to download and experiment with.

14.2 ▪ Bitmaps and the PBM image format

Bitmaps are the simplest of digital images; each pixel is either black or white (or any other color pair). The mathematical representation of the image is a matrix of zeros and ones. *Netpbm* stores a bitmap image to a format called a *portable bitmap* (that's where the name PBM originates) which comes in *plain* and *raw* variants, as noted earlier. By convention, a PBM image file (of either variety) is named with a *.pbm* extension.

The PBM format is not used in the current project; however, I begin the discussion with it since it is so simple and intuitive. There is an elementary application involving images in the PBM format in Chapter 10.

[42]On many Linux platforms the *Netpbm* library may be packaged under the name `libnetpbm10-dev` or something similar. On other platforms you may download the source from `<http://netpbm.sourceforge.net>` and follow the instructions to compile it yourself; it's not difficult.

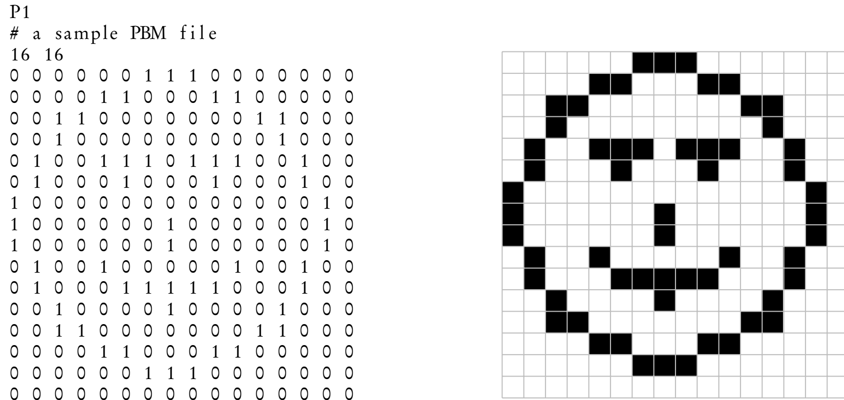


Figure 14.1: On the left are the contents of a plain PBM file. On the right is the corresponding 16×16 bitmap image. I have superimposed a grid to show where the pixels are. Line 2 of the file is a comment.

14.2.1 ▪ The plain PBM format

A *plain PBM* image file consists entirely of ASCII characters. Here is a minimal template for the contents of such a file:

```
P1 w h x x x x x ···
```

The first two characters, `P1`, signal that this is a *plain PBM* file.[43] The w and h are the image's width and height (in pixels). They should be given as ordinary decimal numbers. The `x x x ···` is a sequence of $w \times h$ of zeros and ones. What I have shown as *spaces* between the characters can be one or more *whitespaces* of any kind, i.e., space, tab, newline, etc.; see Section 9.3.2 for the definition of "whitespace". Optionally, a PBM file may contain comments between `P1` and `w`. A comment begins with a "**#**" character and extends to the end of the line. Figure 14.1 shows the contents of a plain PBM file and the corresponding 16×16 bitmap image.

14.2.2 ▪ The raw PBM format

The raw PBM format stores the image according to the template

```
P4 w h -----···
```

where the magic number `P4` identifies the content as a raw portable bitmap, the w and h are image width and height, as before, and each "-" represents a byte of storage. No spaces are allowed between those bytes, and there should be *exactly one whitespace character* between the `h` and the following "-". In modern computer hardware, a byte is typically 8 bits, although this is not a firm requirement. For the purpose of this discussion, let's assume 8-bit bytes. Referring to the bitmap of Figure 14.1, the 16 pixels along the image's

[43] The character sequence `P1` is called the file's *magic number*, although it's not really a number. Magic numbers are employed extensively as file content "signatures". Unix's `file` utility identifies the contents of a given file with some help from the file's magic number. For example,

```
$ file sample-bitmap.pbm
```

prints

```
sample-bitmap.pbm: Netpbm PBM image, ASCII text
```

top row may be encoded in just two bytes, as in

byte 1 `0 0 0 0 0 0 1 1`   byte 2 `1 0 0 0 0 0 0 0`

In contrast, the plain encoding of the `0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0` sequence of 16 zeros, ones, and spaces takes up at least 32 bytes. Therefore, the raw storage format wins by a factor of 16(!) in terms of economy of space.

The wastefulness of the plain PBM format is compensated by its robustness. Quoting from *Netpbm*'s documentation: "It was originally designed to make it reasonable to mail bitmaps between different types of machines using the typical stupid network mailers we have today. Now it serves as the common language of a large family of bitmap conversion filters."

14.3 ▪ Grayscale images and the PGM image format

A bitmap's pixels are either on or off—no shades of gray are possible. A *grayscale image* is like a bitmap in that it maps to a matrix. However, the matrix entries are not limited to zeros and ones; they take values within a prescribed range of integer values from `0` to M (typically $M = 2^8 - 1 = 255$). The magnitude of an entry measures the brightness of the corresponding pixel. Zero is absolutely black, M is maximally white, and the numbers in between correspond to varying shades of gray. *Netpbm* stores a grayscale image in a format called a *portable graymap* (that's where the name PGM originates), which comes in *plain* and *raw* variants, as noted earlier. By convention, a PGM image file (of either variety) is named with a *.pgm* extension.

14.3.1 ▪ The plain PGM format

A *plain graymap* image file consists entirely of ASCII characters. Here is a minimal template for the contents of such a file:

```
P2 w h M x x x x x ···
```

The magic number `P2` signals that this is a *plain PGM* file. The w and h are the image's width and height (in pixels), and M is the image's maximal gray value. They should be given as ordinary decimal numbers. The `x x x ···` is a sequence of $w \times h$ of integers in the range `0` to M inclusive, written as ordinary decimal numbers, with zero being black and M being white. What I have shown as *spaces* between the file's entries can be one or more *whitespaces* of any kind, i.e., space, tab, newline, etc.; see Section 9.3.2 for the definition of "whitespace". Additionally, a PGM file may contain comments between `P2` and `w`. A comment begins with a "`#`" character and extends to the end of the line. Figure 14.2 shows the contents of a plain PGM file and the corresponding 16×16 graymap image. Here M is 15; therefore the gray intensities range from `0` to 15.

14.3.2 ▪ The raw PGM format

The raw PGM format stores the grayscale image according to the template

```
P5 w h M -----···
```

where the magic number `P5` identifies the content as raw PGM, the w and h are image width and height, and M is the image's maximal gray value, as before. If $M < 2^8 = 256$, then each "`-`" represents a byte that encodes the gray value as an 8-digit binary integer in the range `0` to M inclusive. If $2^8 \leq M < 2^{16} = 65536$, then each "`-`" represents a pair

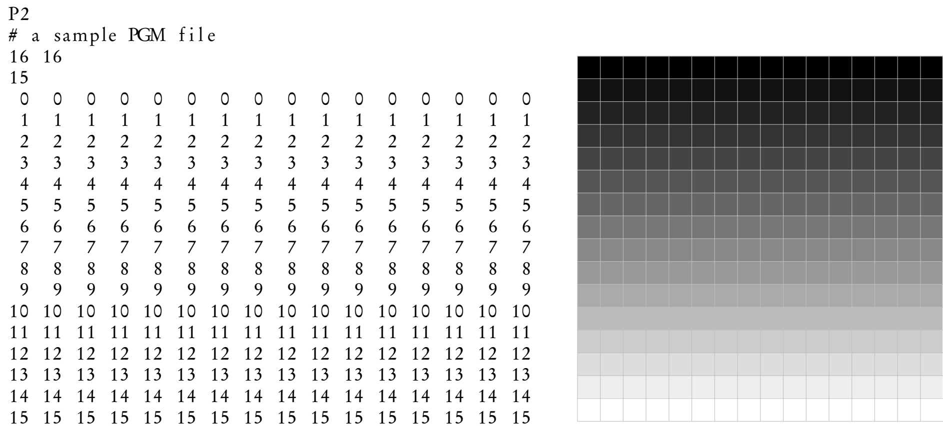


Figure 14.2: On the left are the contents of a plain PGM file. On the right is the corresponding 16×16 graymap image. Gray intensities vary from 0 (black) to 15 (white). I have superimposed a grid to show where the pixels are.

of bytes which together encode the gray value as a 16-digit binary integer in the range 0 to M inclusive. Larger values of M are not permissible. No spaces are allowed between those bytes, and there should be *exactly one whitespace character* between the M and the following "–".

14.4 ▪ Color images and the PPM image format

Within the human eye there are light receptors that are sensitive to red light. There are another type of receptors that are sensitive to green light, and yet another type that are sensitive to blue light. A typical light source emits a mixture of lights of various colors (that is, frequencies). When a light source excites the red and green receptors equally, we perceive a yellow hue. When the intensity of the red and green varies, but their proportions are kept fixed, we perceive darker or brighter yellow. If the light is richer in red than green, we perceive an orange hue.

In general, the colors that we perceive may be broken down to a mixture of *primary colors*, red, green, and blue, of various intensities. An *RGB color image*, also known as a *pixmap image*, is a rectangular array of pixels, where each pixel presents a combination of red, green, and blue colors in varying intensities. The corresponding mathematical model is a matrix of red/green/blue triplets where each part of a triplet takes values in a range 0 to M for some M (typically $M = 2^8 - 1 = 255$). *Netpbm* stores pixmap images in a format called a *portable pixmap* (that's where the name PPM originates), which comes in *plain* and *raw* variants, as in the previous cases. By convention, a PPM image file (of either variety) is named with a *.ppm* extension.

14.4.1 ▪ The plain PPM format

A *plain pixmap* image file consists entirely of ASCII characters. Here is a minimal template for the contents of such a file:

```
P3 w h M r g b r g b ···
```

The magic number `P3` signals that this is a *plain PPM* file. The w and h are the image's width and height (in pixels), and M is the maximal value of the red/green/blue intensities.

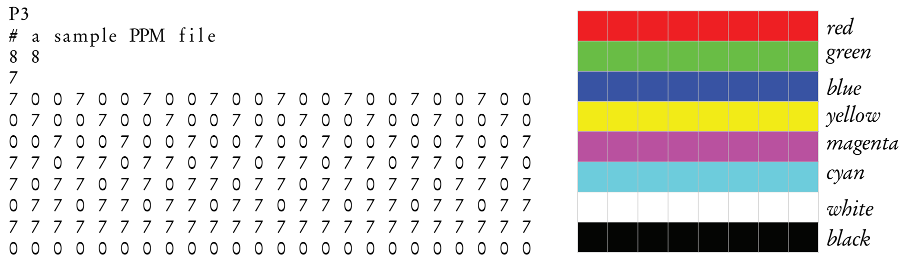


Figure 14.3: On the left are the contents of a PPM file. On the right is the corresponding 8×8 pixmap image. I have superimposed a grid to show where the pixels are. The RGB intensities vary from `0` (black) to `7` (full color). The matrix entries should be read three at a time. The first row consists of eight triplets of the form `{7 0 0}`, which indicates that the top row of pixels are all red. The second row consists of eight triplets of the form `{0 7 0}`, which indicates that the second row of pixels are all green.

They should be given as ordinary decimal numbers. The rest of the template consists of a sequence of $w \times h$ of integer triplets, "`r g b`", where each component is in the range `0` to M inclusive, written as ordinary decimal numbers, with zero indicating an absolutely dim color (black) and M indicating the brightest possible color. What I have shown as *spaces* between the file's entries can be one or more *whitespaces* of any kind, i.e., space, tab, newline, etc.; see Section 9.3.2 for the definition of "whitespace". Additionally, a PPM file may contain comments between `P3` and `w`. A comment begins with a "**`#`**" character and extends to the end of the line. Figure 14.3 shows the contents of a plain PPM file and the corresponding 8×8 pixmap image. Here M is 7; therefore color intensities may range from `0` to `7`, although I have used only `0` and `7` in that sample.

14.4.2 ▪ The raw PPM format

The raw PPM format stores the pixmap image according to the template

```
P6 w h M -----···
```

where the magic number `P6` identifies the content as raw PPM, the w and h are image width and height, and M is the upper range of color values, as before. If $M < 2^8 = 256$, then the rest of the file is a set of $3 \times w \times h$ consecutive bytes. Each byte triplet encodes the red/green/blue colors of a pixel as three 8-digit binary numbers. If $2^8 \le M < 2^{16} = 65536$, then rest of the file is a set of $6 \times w \times h$ consecutive bytes. Each byte sextet encodes the red/green/blue colors of a pixel as three pairs of 16-digit binary numbers. Larger values of M are not permissible. Spaces and comments conform to the same requirements as those of raw PGM files.

14.5 ▪ The *libnetpbm* library

Netpbm's *libnetpbm* library provides C functions for reading and writing PBM, PGM, and PPM images. The interface is declared in the header file *pam.h*, which declares a C structure called "**`struct`** `pam`" for storing all characteristics of an image other than its pixel values. Some of the structure's members deal with issues outside the scope of this presentation. Those members that are relevant to our work shown in Listing 14.1. The meanings of the `file`, `height`, `width` members should be evident from the associated

Listing 14.1: A fragment of *libnetpbm*'s "**struct** pam" structure. Only those parts that are relevant to our work are shown.

```
1  typedef unsigned long sample;
2  struct pam {
3      ...
4      FILE *file;         // the image's input or output stream
5      int format;         // coded value of image type
6      int plainformat;    // boolean: "true" if plain format, "false" if raw format
7      int height;         // image height, in pixels
8      int width;          // image width, in pixels
9      int depth;          // 1 for PBM and PGM, 3 for PPM
10     sample maxval;      // same as M in the previous sections
11     ...
12 };
```

comments. The `format` member is an integer that encodes the image type. It is one of `PBM_FORMAT`, `PGM_FORMAT`, `PPM_FORMAT` in the case of plain image formats or one of `RPBM_FORMAT`, `RPGM_FORMAT`, `RPPM_FORMAT` in the case of raw image formats. The `plainformat` member is a Boolean variable which is *true* (i.e., 1) if the image format is plain and *false* (i.e., 0) if the image format is raw. Considering that the `format` member already carries the information regarding raw versus plain, the `plainformat` member appears to be redundant. In a sense, it is—it exists as a hack to address compatibility issues with the earlier versions of *libnetpbm*. When writing an image to a file, the latest versions of the library ignore the raw versus plain information contained in the `format` member and use the raw format as a default. To get a plain format in the output, you will need to set `plainformat` to *true*.

The `depth` member is set to 1 for PBM and PGM images because these require a single number to specify a pixel value. It is set to 3 for a PPM image because it requires 3 numbers to specify a pixel value.

The `maxval` member is what I called M in the previous section. Pixel values are integers in the range $0, 1, \dots, M$. The `maxval` is declared as an object of type "`sample`". We see on line 1 that this is an alias for **`unsigned long`** in *libnetpbm*.

Libnetpbm provides a multitude of functions for manipulating image objects. In the rest of this section I will describe those functions—10, to be precise—that we need in our work. The descriptions are terse but should be adequate for most purposes. For more details you should consult *libnetpbm*'s documentation.

```
void pm_init(const char *progname, unsigned int flags);
```

Every *libnetpbm* program should call `pm_init()` to initialize the library. The first argument is the program's name. The second argument is always zero. (It's really a placeholder for possible future expansions of the library.)

```
FILE *pm_openr(char *name);
FILE *pm_openw(char *name);
FILE *pm_close(FILE *fp);
```

These are the fancy versions of the standard library's `fopen()` and `fclose()` functions. Specifically, `pm_openr()` and `pm_openw()` open a file for reading or writing, respectively, and call `exit()` if something goes wrong. The file name `"-"` is interpreted as the `stdin` or `stdout`, as the case may be. The function `pm_close()` closes a previously opened stream.

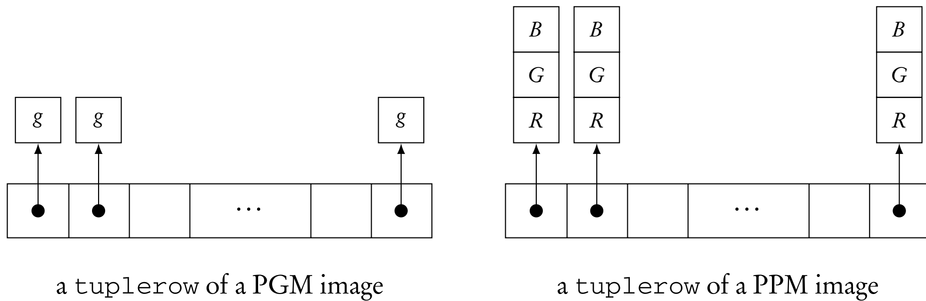


Figure 14.4: A tuplerow array holds the pixel data of one row of an image. The array's length equals the image's width. The array's entries point to tuple vectors that hold a pixel's gray values in the case of a PGM image (shown on the left) or the RGB values in the case of PPM image (shown on the right). The tuple vectors of a PGM image are of length 1. Those of a PPM image are of length 3.

```
void pnm_readpaminit(FILE *fp, struct pam *pam, int size);
void pnm_writepaminit(struct pam *pam);
```

The function `pnm_readpaminit()` reads an image's header information[44] from the stream `fp` and populates the `pam` structure with it. It expects to find the stream positioned at the start of the image's header and leaves it positioned at the start of the image's data.

The meaning of the `size` argument is a bit too complex to explain here, and probably an explanation won't be very illuminating. Suffice it to say that you should always pass the expression `PAM_STRUCT_SIZE(tuple_type)` for that argument. This assumes that you have a relatively recent *libnetpbm* library. In the older versions of the library (specifically, versions before *Netpbm* 10.23) pass `sizeof(struct pam)` instead.

The function `pnm_writepaminit()` writes an image's header, according to the information supplied in the `pam` structure, to the output stream associated with the image.

```
tuple *pnm_allocpamrow(struct pam *pam);
void pnm_freepamrow(tuple *tuplerow);
```

These functions allocate and free memory for storing one row of an image's pixel data into an array called a `tuplerow`, shown schematically in Figure 14.4. The array's length equals the image's width. Each entry of the `tuplerow` is a pointer to an array, called a `tuple` array. In the case of a PGM image, a `tuple` array has length 1. It holds the pixel's gray value. In the case of a PPM image, a `tuple` array has length 3. It holds the pixel's red, green, and blue values, in that order.

```
void pnm_readpamrow(struct pam *pam, tuple *tuplerow);
void pnm_writepamrow(struct pam *pam, const tuple *tuplerow);
```

The function `pnm_readpamrow()` reads one row of an image into a `tuplerow` array (see the previous paragraph). It expects to find the input stream associated with the image to be positioned at the start of the desired row and leaves it positioned just after it.

The function `pnm_writepamrow()` writes one row of an image from a `tuplerow` array. It expects to find the output stream associated with the image to be positioned where the row should start and leaves it positioned just after it.

[44] The *image header* is everything in the image file other than the pixel data section.

14.6 ▪ A no-frills demo of *libnetpbm*

The file *netpbm-minimal.c* shown in Listing 14.2 provides a no-frills demo of *libnetpbm*. It reads a PGM image into a matrix a, draws a black line over it by changing the pixels a_{ii}, $i = 0, 1, 2, \ldots$, to zero, and then writes the result into another PGM file. If you have read the descriptions of the *libnetpbm* functions in the previous section, you should have no difficulty in following the program's logic and understanding what it does; therefore I will not elaborate on it any further other than pointing out that on line 42 we set the `pam` structure's `plainformat` member to *true* or *false*, according to the input image's format, so that a plain format input results in a plain format output and a raw format input results in a raw format output. In the absence of that line, the output image will be in the raw format regardless of the input image's format because that is *libnetpbm*'s default behavior.

The program depends on Chapter 8's *array.h*, which in turn depends on Chapter 7's *xmalloc* module. Therefore I suggest that you construct a *Makefile* according to the instructions of Chapter 6 to build *netpbm-minimal* as well as the several other programs which you will develop in this chapter. Here is the beginning part of the *Makefile*:

```
CFLAGS = -Wall -pedantic -std=c99 -O2
LIBS   = -lnetpbm
netpbm-minimal: xmalloc.o netpbm-minimal.o
	$(CC) $^ $(LIBS) -o $@
```

14.7 ▪ The interface of the *image-io* module

We are going to develop an *image-io* module which will provide functions for reading and writing PGM and PPM images. Listing 14.3 shows the file *image-io.h* that provides the module's interface. It declares a "**struct** image", which extends *libnetpbm*'s "**struct** pam" by adding new members `r`, `g`, `b` that point to $h \times w$ matrices that hold an image's red, green, and blue pixel values. (Here h and w are the image's height and width in pixels.) Additionally, it declares the prototypes of functions that read and write images and free the allocated memory.

We use "**struct** image" with both PGM and PPM image types. In the case of a PGM image only one of the three matrix pointers `r`, `g`, `b` is needed. I will use the `g` pointer consistently for that purpose since "g" may be thought of as standing both for "green" and "gray". The `r` and `b` pointers are not used in that case and may be left unassigned.

Remark 14.4. Why are the `r`, `g`, `b` matrices in "**struct** image" of type **double**? The pixel values of an image are integers in the range 0 to M, where typically M is $2^8 - 1$. Isn't it more natural to store the numbers in a matrix of type **unsigned char**? Even in the worst case, when M is $2^{16} - 1 = 65535$), a matrix of type **unsigned short int** ought to suffice.

Those observations are valid. Nevertheless, the type of storage for an image is dictated by the application. I am using the type **double** for storage in anticipation of applications of the *wavelet transform* to image analysis, which is the subject of the next chapter. There we overwrite an image matrix with the corresponding wavelet transform, which is no longer a matrix of integers. If your applications don't do such drastic things to your images, then feel free to change **double** to **unsigned char** or **unsigned short int** in *image-io.h*.

The *image-io* module relies on the *xmalloc* module (Chapter 7) for memory allocation and the *array.h* header file (Chapter 8) for building matrices. Therefore, following the suggestions in Chapter 2, the project's directory will contain

Listing 14.2: The file *netpbm-minimal.c* provides a no-frills demo of the *libnetpbm* functions. It reads a PGM image into a matrix, draws a black line over it, and then writes the result into another PGM file. Calls to the *libnetpbm* functions are set on a shaded background.

```
1  #include <stdio.h>
2  #include <pam.h>
3  #include "array.h"
4  int main(int argc, char **argv)
5  {
6      char *infile = "aya_matsuura.pgm";
7      char *outfile = "zz.pgm";
8      struct pam pam;
9      FILE *fp;
10     double **a;
11     tuple *row;
12     int i, j;
13
14     pm_init(argv[0], 0);                       // initialize the library
15
16     // read image header
17     fp = pm_openr(infile);
18     pnm_readpaminit(fp, &pam, PAM_STRUCT_SIZE(tuple_type));
19     printf("image: %dx%d, depth: %d\n",
20             pam.width, pam.height, pam.depth);
21     if (!(pam.format == PGM_FORMAT || pam.format == RPGM_FORMAT)) {
22         fprintf(stderr, "Sorry, this demo handles PGM images only.\n");
23         return EXIT_FAILURE;
24     }
25
26     make_matrix(a, pam.height, pam.width);   // matrix to store image data
27     row = pnm_allocpamrow(&pam);              // tuplerow vector
28     for(i = 0; i < pam.height; i++) {         // read pixel data
29         pnm_readpamrow(&pam, row);
30         for (j = 0; j < pam.width; j++) {    // copy tuplerow into matrix
31             a[i][j] = row [j][0];
32         }
33     }
34     pm_close(fp);
35
36     for (i = 0; i < pam.width && i < pam.height; i++)
37         a[i][i] = 0.0;                        // change the a_ii pixels to black
38
39     // write image header
40     fp = pm_openw(outfile);
41     pam.file = fp;
42     pam.plainformat = pam.format == PGM_FORMAT ? 1 : 0;
43     pnm_writepaminit(&pam);
44
45     for (i = 0; i < pam.height; i++) {        // write pixel data
46         for(j = 0; j < pam.width; j++)
47             row[j][0] = a[i][j];
48         pnm_writepamrow(&pam, row);
49     }
50     pm_close(fp);
51     free_matrix(a);
52     pnm_freepamrow(row);
53     return EXIT_SUCCESS;
54 }
```

Listing 14.3: The file *image-io.h*.

```
1  #ifndef H_IMAGE_IO_H
2  #define H_IMAGE_IO_H
3  #include <pam.h>
4  struct image {
5      struct pam pam;
6      double **r;
7      double **g;
8      double **b;
9  };
10 struct image *read_image(char *filename);
11 void write_image(char *filename, struct image *img);
12 void free_image(struct image *img);
13 #endif /* H_IMAGE_IO_H */
```

```
$ cd image-io
$ ls -F
Makefile            image-io-test-3.c  netpbm-minimal.c
array.h@            image-io-test-4.c  xmalloc.c@
image-io-test-0.c   image-io-test-5.c  xmalloc.h@
image-io-test-1.c   image-io.c
image-io-test-2.c   image-io.h
```

The stand-alone file *netpbm-minimal.c* was presented in Section 14.6. The *image-io* module itself is contained in the files *image-io.[ch]*. The files *image-io-test-[0–5].c* are independent drivers for demonstrating the module's functionality in various ways, as will be described later. They compile to executables named *image-io-test-[0–5]*. Each executable is invoked with two arguments, as in

```
$ image-io-test-0 infile outfile
```

The program reads an image from *infile*, modifies the images in certain ways, and writes the modified image to `outfile`. You may name the input and output files as you wish, but it is customary to name images in PGM and PPM formats with the *.pgm* and *.ppm* extensions, respectively. The program does not rely on a file name to determine the image format. Rather, it detects the image format through the magic number embedded in the file.

Remark 14.5. In the descriptions of *libnetpbm*'s `pm_openr()` and `pm_openw()` functions in Section 14.5, we noted that these functions treat the special file name `"-"` as a reference to the `stdin` or `stdout`. Therefore, the program may also be invoked as

```
$ image-io-test - outfile.pgm <infile.pgm
```

or even as

```
$ image-io-test - - <infile.pgm >outfile.pgm
```

These will work in Unix or in Unix-like systems where a file is treated a stream of bytes. On an operating system such as Windows that distinguishes between text and binary files, the redirection may corrupt a raw image.

Listing 14.4: An outline of the file *image-io.c*.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "array.h"
4  #include "image-io.h"
5  ▶ static void read_pgm_pixel_data(struct image *img) ...
6  ▶ static void read_ppm_pixel_data(struct image *img) ...
7  ▶ struct image *read_image(char *filename) ...
8  ▶ static void write_pgm_pixel_data(struct image *img) ...
9  ▶ static void write_ppm_pixel_data(struct image *img) ...
10 ▶ void write_image(char *filename, struct image *img) ...
11 ▶ void free_image(struct image *img) ...
```

Listing 14.5: An implementation of the function `read_image()`.

```
1  struct image *read_image(char *filename)
2  {
3      struct image *img = xmalloc(sizeof *img);
4      struct pam *pam = &img->pam;
5      FILE *fp = pm_openr(filename);
6      pnm_readpaminit(fp, pam, PAM_STRUCT_SIZE(tuple_type));
7      if (pam->format == PGM_FORMAT || pam->format == RPGM_FORMAT)
8          read_pgm_pixel_data(img);
9      else if (pam->format == PPM_FORMAT || pam->format == RPPM_FORMAT)
10         read_ppm_pixel_data(img);
11     else {
12         fprintf(stderr, "error: file '%s' contains neither "
13                 "a PGM nor a PPM image\n", filename);
14         exit(EXIT_FAILURE);
15     }
16     pm_close(fp);
17     return img;
18 }
```

14.8 ▪ The implementation of the *image-io* module

Listing 14.4 shows an outline of the file *image-io.c*. I will give the details of the parts dealing with PGM images. You will write the parts dealing with PPM images.

14.8.1 ▪ The function `read_image()`

The function `read_image()` that appears on line 7 of Listing 14.4 receives the name of an existing image file. It reads the file's contents, allocates memory for a "`struct image`" structure and its members, populates the structure with the image's data, and returns a pointer to that structure. If anything fails in the process, it prints a diagnostic message to the `stderr` and calls `exit()` to kill the program. Listing 14.5 shows my implementation of `read_pgm_image()`. It handles PGM and PPM images, both the plain and raw types. Extending it to handle PBM images is straightforward, and you are welcome to try it yourself, but I will refrain from doing so in the interest of brevity.

Let us go through the details of the function `read_image()`.

Line 3. We allocate memory for a "**struct** image" and set `img` to point to that memory location. The purpose of the current function is to read image data from the input file, deposit that information into that structure, and return the structure's address to the caller.

Line 4. The pointer `&img→pam` is needed in quite a few places in the rest of the code. It wouldn't be wrong to write `&img→pam` everywhere, but it would clutter things. The `pam` variable defined here is a shorthand meant to reduce clutter; doing so isn't essential in any way.

Line 5. Here we call *libnetpbm*'s `pm_openr()` function, described in Section 14.5, to open the image file for reading. We could have called C's standard `fopen()` instead, as in

```
FILE *fp = fopen(filename, "r");
```

but, as noted in Section 14.5, `pm_openr()` offers a couple of extra helpful features. For one thing, it treats the file name `"-"` as a request for `stdin`, as seen in the usage examples in Remark 14.5 on page 123. For another thing, it checks for errors. If the requested file cannot be opened for reading, it prints a message to `stderr` and exits the program! Consequently, there is no point in checking for the success of the `pm_openr()` call.

Line 6. Here we call *libnetpbm*'s `pnm_readpaminit()` function, described in Section 14.5, to read the image's header data. The first argument is the stream `fp`, which was set up on the previous line. The second argument, `pam`, points to the "**struct** pam", which is embedded in "**struct** image". The function will populate `pam`'s members (see Listing 14.1) with the information that it extracts from the image's header. The third argument, as noted in Section 14.5, is always the expression I have shown. Also as was noted there, you may have to change it to `sizeof(struct pam)` if you have an older version of *libnetpbm*.

Line 7. Here the program branches depending on the image type. The `format` member of "**struct** pam" encodes the image type that has just been read from the input stream (see Section 14.5 for the details of the "**struct** pam"). If the image is of type PGM, we call `read_pgm_pixel_data()` to read the pixel data. If the image is of type PPM, we call `read_ppm_pixel_data()` to read the pixel data. (These functions will be described shortly.) If the image is something else—presumably a PBM or a PAM[45]—then we print a message saying that we are not prepared to handle that type and exit the program.

Lines 16 and 17. We are done with reading the image; therefore we close the input stream and return `img` to the caller. The function `pm_close()`, described in Section 14.5, is a wrapper around C's standard `fclose()`—it checks for `fclose()`'s return status and calls `exit()` if an error occurs.

[45] I have neither mentioned the PAM format up to this point, nor will I deal with it from here on. PAM is an "adaptable" format in the sense that any PBM, PGM, or PPM image may be presented as a PAM image. You will have to read *Netpbm*'s documentation if you want to learn more about that format.

Listing 14.6: The implementation of the function `read_pgm_pixel_data()`.

```
1  static void read_pgm_pixel_data(struct image *img)
2  {
3      struct pam *pam = &img→pam;
4      tuple *row = pnm_allocpamrow(pam);
5      make_matrix(img→g, pam→height, pam→width);
6      img→r = img→b = NULL;
7      for (int i = 0; i < pam→height; i++) {
8          pnm_readpamrow(pam, row);
9          for (int j = 0; j < pam→width; j++)
10             img→g[i][j] = row[j][0];
11     }
12     pnm_freepamrow(row);
13 }
```

14.8.2 ▪ The function `read_pgm_pixel_data()`

The function `read_pgm_pixel_data()` that appears on line 5 of Listing 14.4 reads the pixel data of a PGM image. Its implementation is shown in Listing 14.6. Let us go through the details.

Line 3. The variable `pam` is introduced here for the same reason as that on line 4 of Listing 14.5. It's not essential in any way; it's a shorthand meant to reduce clutter.

Line 4. Here we call *libnetpbm*'s `pnm_allocpamrow()` function, described in Section 14.5, to allocate memory for a `tuplerow` array. We are going to read the image one row at a time into that array and then copy the result into a matrix. Figure 14.4 shows a schematic representation of a `tuplerow` array. There is no need to check for success—if memory allocation fails, `pnm_allocpamrow()` will print a message to `stderr` and exit the program.

Lines 5 and 6. We call the `make_matrix()` macro from the header file *array.h* (see Chapter 8) to allocate a matrix of **`double`**, assigned to `img→g`, to hold the pixel data of a `(pam→height)` × `(pam→width)` image. The members `img→r` and `img→b` are meant for use in PPM images, so they are not needed here. We set them to `NULL` so that they can be passed to `free_matrix()` with no ill effects; see subsection 14.8.5.

Lines 7–11. The **`for`**-loop goes through the image one row at a time. At each step it calls *libnetpbm*'s `pnm_readpamrow()` function, described in Section 14.5, to read an entire row into the `row` array. Thus, `row[j]` points to the tuple associated with the pixel at row `i` and column `j`. Since this is a PGM image, the tuple vector has length 1 (see Figure 14.4); therefore its first (and only) element is given by `row[j][0]`. We store that value in `img→g[i][j]`.

Line 12. We are done with reading. We call *libnetpbm*'s `pnm_freepamrow()` function, described in Section 14.5, to free the memory allocated on line 4.

14.8.3 ▪ The function `write_image()`

The function `write_image()` that appears on line 10 of Listing 14.4 receives the name of an output file and a pointer to a fully populated "**`struct`** `image`". It writes the

Listing 14.7: The implementation of the function `write_image()`.

```
1  void write_image(char *filename, struct image *img)
2  {
3      struct pam *pam = &img→pam;
4      pam→file = pm_openw(filename);
5      pam→plainformat
6          = (pam→format == PGM_FORMAT || pam→format == PPM_FORMAT) ? 1 : 0;
7      pnm_writepaminit(pam);
8      if (pam→format == PGM_FORMAT || pam→format == RPGM_FORMAT)
9          write_pgm_pixel_data(img);
10     else if (pam→format == PPM_FORMAT || pam→format == RPPM_FORMAT)
11         write_ppm_pixel_data(img);
12     else {
13         fprintf(stderr, "error: file %s, line %d: shouldn't be here!\n",
14                 __FILE__, __LINE__);
15         exit(EXIT_FAILURE);
16     }
17     pm_close(pam→file);
18 }
```

image's data into the named file as a PGM or PPM image. If a file of the given name does not exist, it's created. If it exists, it's overwritten. If anything fails in the process, it prints a diagnostic message to the `stderr` and exits the program. Listing 14.7 shows the implementation of `write_image()`.

Line 3. We introduce the variable `pam` as a shorthand for `&img→pam`, just as we did in `read_image()`. This is not essential; it serves only to reduce clutter.

Line 4. Here we call *libnetpbm*'s `pm_openw()` function, described in Section 14.5, to open the image file for writing. We could have called C's standard `fopen()` instead, as in

```
FILE *fp = fopen(filename, "w");
```

but, as noted in Section 14.5, `pm_openw()` offers a couple of extra helpful features. For one thing, it treats a file named `"-"` as a request for `stdout`; see the usage examples in Remark 14.5 on page 123. For another thing, it checks for errors. If the requested file cannot be opened for writing, it prints a message to `stderr` and exits the program! Consequently, there is no point in checking for the success of the `pm_openw()` call.

Note that we set `pam→file` to the stream opened by `pm_openw()`. Other functions which write to the stream look up the stream's address in the `pam` structure.

Line 5. As noted in Section 14.5, *libnetpbm* ignores the format encoded in the `pam` structure's `format` member and writes images in the raw format by default. Here we examine the `format` member. If it indicates a plain format, we set the `plainformat` member to *true* to override the library's default behavior.

Line 7. *Libnetpbm*'s `pnm_writepaminit()` function, described in Section 14.5, writes an image's header data (e.g., the "`P5 w h M`" part described in subsection 14.3.2) by extracting the needed information from the `pam` structure. The output goes to the stream `pam→file`.

Listing 14.8: The implementation of the function `write_pgm_pixel_data()`.

```
1  static void write_pgm_pixel_data(struct image *img)
2  {
3      struct pam *pam = &img→pam;
4      tuple *row = pnm_allocpamrow(pam);
5      for (int i = 0; i < pam→height; i++) {
6          for (int j = 0; j < pam→width; j++)
7              row[j][0] = img→g[i][j];
8          pnm_writepamrow(pam, row);
9      }
10     pnm_freepamrow(row);
11 }
```

Line 8. Here the program branches depending on the image type. If the image is of type PGM, we call `write_pgm_pixel_data()` to write the pixel data. If the image is of type PPM, we call `write_ppm_pixel_data()` to write the pixel data. (These functions will be described shortly.) Unless we have carelessly meddled with the `pam` structure's `format` member, these two should be the only possibilities. Anything else is abnormal and should never occur. Nevertheless, we supply code to catch the hypothetical implausible case, write a diagnostic, and exit the program.

Line 17. We are done with writing; therefore we call `pm_close()`, described in Section 14.5, to close the output stream. The function `pm_close()` is a wrapper around C's standard `fclose()`—it checks for `fclose()`'s return status and calls `exit()` if an error occurs.

14.8.4 ▪ The function `write_pgm_pixel_data()`

The function `write_pgm_pixel_data()` that appears on line 8 of Listing 14.4 has many aspects in common with `read_pgm_pixel_data()`; therefore my comments will be somewhat terser here. It receives a pointer to a fully populated "**`struct`** `image`" corresponding to a PGM image and writes the image's pixel data into an output stream. Its implementation is shown in Listing 14.8. Let us go through the details.

Lines 3 and 4. These are just like what we had in `read_pgm_pixel_data()`. Read the explanations there.

Lines 5–9. The outer **`for`**-loop goes through the `img→g` matrix one row at a time. The inner **`for`**-loop goes through the row and copies the value of `img[i][j]` to the first (and only) element of the tuple associated with the pixel `row[j]`. (This is the opposite of what we did when reading the image.)

Remark 14.6. Note that `img→g[i][j]` is of type **`double`**, while `row[j][0]` is of type "`sample`", which is an alias for **`unsigned long`**. C's automatic conversion rules are applied during the assignment. If the right-hand side is an integer stored as a **`double`**, then the left-hand side receives the exact value of that integer. If the right-hand side is a noninteger, then its fractional part is dropped! For instance, both 3.1 and 3.9 are converted to 3; no rounding takes place. It is the responsibility of the caller to do all necessary rounding/processing on the image's sample values before arriving here. In particular, it is the caller's responsibility to ensure that the

Listing 14.9: The implementation of `free_image()` in file *image-io.c*.

```
1  void free_image(struct image *img)
2  {
3      if (img ≠ NULL) {
4          free_matrix(img→r);
5          free_matrix(img→g);
6          free_matrix(img→b);
7          free(img);
8      }
9  }
```

sample values lie in the range 0 to M, where M is the image's maximum sample value. In Chapter 15's applications of wavelets to image analysis, we take care of these details before `write_pgm_pixel_data()` is called. This is the purpose of the `clip_matrx()` function introduced in subsection 15.6.2.

Line 8. The function `pnm_writepamrow()`, described in Section 14.5, writes the pixel data for an entire row of an image to the output stream.

Line 10. Here we free the memory associated with the `row` vector which was previously allocated on line 4.

14.8.5 ▪ The function `free_image()`

The function `free_image()` that appears on line 11 of Listing 14.4 (page 124) frees memory resources associated with a "**struct** image". The implementation shown in Listing 14.9 is quite straightforward. It frees the following:

- The memory associated with the matrices `img→r`, `img→g`, `img→b`, allocated in `read_pgm_pixel_data()` or `read_ppm_pixel_data()`; see line 5 of Listing 14.6 on page 126.
- The memory associated with the `img` structure, allocated in `readimage()`; see line 3 of Listing 14.5 on page 124.

As with all other `free_*()` functions in this book, as well as the standard library's `free()`, calling `free_image()` with a `NULL` argument is safe.

Remark 14.7. Note the significance of `NULL`-ing the `img→r` and `img→b` pointers on line 6 in Listing 14.6. Our `free_matrix()` in *array.h* is implemented to handle a `NULL` argument gracefully.

14.9 ▪ The file *image-io-test-0.c*

The file *image-io-test-0.c*, shown in Listing 14.10, provides a driver for exercising the *image-io* module. It simply reads a PGM or PPM image (plain or raw format) and writes it out without change, except for embedded comments, if any, which are discarded. The names of the input and output files are obtained from `argv[1]` and `argv[2]`. Thus, it is invoked as

```
$ image-io-test-0 infile outfile
```

Listing 14.10: The file *image-io-test-0.c*, shown in its entirety here, provides a driver for exercising the *image-io* module. It reads a PGM or PPM file and writes it out without change, except for embedded comments, if any, which are discarded. The names of the input and output files are obtained from `argv[1]` and `argv[2]`.

```
1  #include <stdio.h>
2  #include "image-io.h"
3  int main(int argc, char **argv)
4  {
5      if (argc ≠ 3) {
6          fprintf(stderr, "Usage: %s infile outfile\n", argv[0]);
7          fprintf(stderr, "Reads PGM or PPM from infile, "
8                          "writes identical image to outfile\n");
9          return EXIT_FAILURE;
10     }
11     pm_init(argv[0], 0);
12     struct image *img = read_image(argv[1]);
13     fprintf(stderr, "image is %dx%d, depth=%d, maxval=%d\n",
14             img→pam.height, img→pam.width,
15             img→pam.depth, (int)img→pam.maxval);
16     write_image(argv[2], img);
17     free_image(img);
18     return EXIT_SUCCESS;
19 }
```

See Section 14.7 regarding the invocation details. Here is a line-by-line description of *image-io-test-0.c*:

Line 5. We expect the program to be invoked with exactly two arguments, Thus, `argc` should be 3 (see Section 4.7). Then `argv[0]` is the program's name, `argv[1]` is the name of the input image file, and `argv[2]` is the name of the output image file. If `argc` is other than 3, we print a message to `stderr` and exit.

Line 11. As noted in Section 14.5, every *libnetpbm* program is expected to call `pm_init()` to initialize the library. The first argument of `pm_init()` should be the name of the program. The second argument is always zero.

Line 12. The function `read_image()` (see subsection 14.8.1) reads the image file corresponding to `argv[1]`. When it returns, everything about the image is stored in the `img` structure.

Note the midblock declaration of **`struct`** `image`. This is permissible in C99 but not in C89. If you wish to reduce the code to C89, then insert a declaration **`struct`** `image *img;` at the top of the block and change the current line to `img = read_image(argv[1]);`.

Line 13. We print some of the image's data as a feedback to the user. The value of `depth` will be 1 in the case of a PGM image and 3 in the case of a PPM image.

Line 16. The function `write_image()` (see subsection 14.8.3) writes the image from the `img` structure into the file corresponding to `argv[2]`.

Line 17. We call the function `free_image()` (see subsection 14.8.5) to free all memory resources associated with the image and then exit.

14.10 ▪ *Project Image I/O*

Part 14.1. Complete, compile, and test the *netpbm-minimal* program of Section 14.6.

Part 14.2. In the preceding sections I gave the complete code for reading and writing PGM images. Complete the *image-io* module by adding functionality to handle PPM images. This requires implementing the functions `read_ppm_pixel_data()` and `write_ppm_pixel_data()`; see lines 6 and 9 of Listing 14.4 on page 124. Modifications relative to the PGM image handling code are very slight. Figure 14.4 on page 120 should provide all the hints you need to get started.

Part 14.3. Compile and test your *image-io-test-0* using some of the PGM and PPM images that I have provided in the book's website.[46] For instance,

```
$ ./image-io-test-0 aya_matsuura.pgm z0.pgm
image is 512x512, depth=1, maxval=255
```

The program reads the image file *aya_matsuura.pgm* and writes the new image file *z0.pgm*. Since we have not modified the image, the two files should be identical. You may apply the Unix utility called `cmp` to verify that the files are indeed identical:

```
$ cmp aya_matsuura.pgm z0.pgm
```

If the files being compared are different, `cmp` prints the location of the first byte where a difference occurs. If the files are identical, `cmp` exists silently.

Remark 14.8. It is *not true* in general that the input and output files of *image-io-test-0* will be identical. In the description of image file formats—see, e.g., subsection 14.3.2—I noted that an image's header material may be interspersed with optional comments and whitespace. Our `read_image()` function discards header comments as they are read. Therefore two identical images need not have identical files since their header sections may be different. Nevertheless, the image files in the book's website contain no comments (except for the one noted explicitly as such), and the whitespace in their headers is according to what *libnetpbm*'s utilities create; therefore your *aya_matsuura.pgm* and *z0.pgm should be identical*. If not, your program is buggy.

Part 14.4. [optional] Write a program file, *image-io-test-1.c*, that converts a (color) PPM image to a (grayscale) PGM image. I suggest that you start with a copy of *image-io-test-0.c* shown in Listing 14.10. Between reading the image (line 12) and writing the image (line 16) insert

```
img->pam.format = RPGM_FORMAT;
img->pam.depth = 1;
```

This sets the image's output format to raw PGM and the image's depth to 1 because a PGM image's pixel values are identified by a single number. Now, loop over the image, and at each pixel do

```
double avg = (img->r[i][j] + img->g[i][j] + img->b[i][j])/3.0;
img->g[i][j] = (int)(0.5 + avg);
```

On the first line, we compute the *average* of the pixel's red, green, and blue values. The average is not necessarily an integer; however, a PGM image expects integers for its pixel

[46] See Remark 14.3 on page 114.

values. Therefore, on the second line we cast the result to an **int** before assigning it to img→g[i][j]. Note that casting a **double** to **int** discards the number's fractional part. Adding the 0.5 ensures proper rounding. For instance, if avg is 3.1, then $0.5+3.1$ is 3.6, which, after discarding the fractional part, is reduced to 3. On the other hand, if avg is 3.9, then $0.5+3.9$ is 4.4, which, after discarding the fractional part, results in 4. You will find an extended discussion of this in subsection 15.6.2 on page 141.

Your program will expect to be invoked with two arguments, as in

```
$ ./image-io-test-1 infile outfile
```

where infile is expected to be a PPM image. It should complain if it is invoked with the wrong number of arguments or if the infile is not a PPM image. Here is what mine does:

```
# invoked with too few args
$ ./image-io-test-1
Usage: ./image-io-test-1 infile outfile
Reads PPM image from infile, writes averaged PGM image to outfile

# infile is not a PPM image
$ ./image-io-test-1 aya_matsuura.pgm z1.pgm
./image-io-test-1: expected a PPM image in input

# a normal run
$ ./image-io-test-1 aya_matsuura.ppm z1.pgm
input image is 512x512, depth=3, maxval=255
```

Part 14.5. [optional] As natural as the straight averaging of the RGB values may seem, it does not work quite well. The fact is, the human eye is not equally sensitive to the luminosity of the red, green, and blue colors. Experience shows that a weighted average, with a larger weight attached to the green component, produces more natural-looking results. *Libnetpbm*'s conversion routines mix the red, green, and blue in the 77:150:29 proportions, although this is not a hard and fast rule.

Copy *image-io-test-1.c* to *image-io-test-2.c*, and, noting that $77+150+29=256$, change the averaging formula to

```
double avg = (77*img→r[i][j] + 150*img→g[i][j]
               + 29*img→b[i][j])/256.0;
```

Apply the program to convert the previous PPM image into a file *z2.pgm*. View and compare the images *z1.pgm* and *z2.pgm*.

Part 14.6. [optional] The image viewing program *Xv* averages the red, green, and blue colors in the 20:32:12 proportions. Copy *image-io-test-2.c* to *image-io-test-3.c*, and change the averaging algorithm according to this. Apply the program to convert the previous PPM image into a file *z3.pgm*. View and compare the images *z1.pgm*, *z2.pgm*, and *z3.pgm*.

Part 14.7. [optional] Copy *image-io-test-0.c* to *image-io-test-4.c*. Modify it so that it "rotates" the colors of a PPM image. That is, it writes the red colors to the green matrix, the green colors to the blue matrix, and the blue colors to the red matrix. View the resulting image.

Part 14.8. [optional] Write a program file, *image-io-test-5.c*, that draws a black "X" over an image's diagonals. In an image of height h and width w, the diagonal D_1 that goes from

the northwest to the southeast is given by

$$D_1 = \{(i,j) : \quad i = 0,\ldots,h-1, \quad j = i(w-1)/(h-1)\}.$$

Observe that when $i = 0$, we get $j = 0$, and when $i = h-1$, we get $j = w-1$, as expected. Note, however, that since i, h, and w are integers, the proper C encoding of the formula is

```
int j = 0.5 + (double)i*(w-1)/(h-1);
```

You figure out how to do the other diagonal. See Part 14.4 for an explanation of the extra 0.5.

Chapter 15
Image analysis

Prerequisites: Chapters 7, 8, 13, 14

15.1 ▪ Introduction

In Chapter 14 we saw how to import a PGM or PPM image as a matrix of pixels. In Chapter 13 we saw how to apply the Haar wavelet transform to express a matrix in terms of the Haar wavelet basis. In this chapter we bring these techniques to their natural conclusion by applying the Haar wavelet transform to an image. This decomposes the image into a sum of progressively finer wavelets. The very fine wavelets tend to have very small coefficients because they measure the differences between the image's nearby pixels. In an ordinary photograph, say of a person's face, the differences between adjacent pixels are small, in general, because the texture and coloration vary smoothly. The cheeks, chin, and the forehead tend to be wide expanses of slowly varying pixels. There are exceptions, however. For instance, a finely knitted sweater, or distinctively combed hair, may exhibit significant variations even at the pixel scale. In any case, even in images containing fine textures, most of the coefficients of the finer wavelets tend to be very small. Figure 15.1 shows a 512×512 grayscale image[47] on the right. The two graphs on the left show the absolute values of the image's wavelet coefficients, sorted in decreasing order in size. Thus, the horizontal axis enumerates the coefficients (these go from 1 to $512 \times 512 = 262{,}144$) and the vertical axis is their magnitudes. The graph on the top shows the full range of the over quarter million coefficients. You can hardly see anything there because essentially all the coefficients are practically zero, and thus the graph coincides with the horizontal axis. If you look *very closely*, you may discern a slight rounding near the origin. That's all there is to be seen of the graph. In the bottom graph I have expanded the horizontal range $1, \ldots, 80$ of the upper graph to zoom into the details near the origin. The very first coefficient is approximately 115. The next few are approximately 42, 41, 21, 16, 14, and they drop rapidly after that.

This raises an intriguing question: What if we keep the first few thousand significant wavelet coefficients and discard the rest? Reconstructing the image from the retained coefficients should yield a relatively faithful representation of the image since the discarded ones are too small to matter.

[47] Actually the image that you see in the book has been reduced to fit the page. The actual image I analyzed was 512×512.

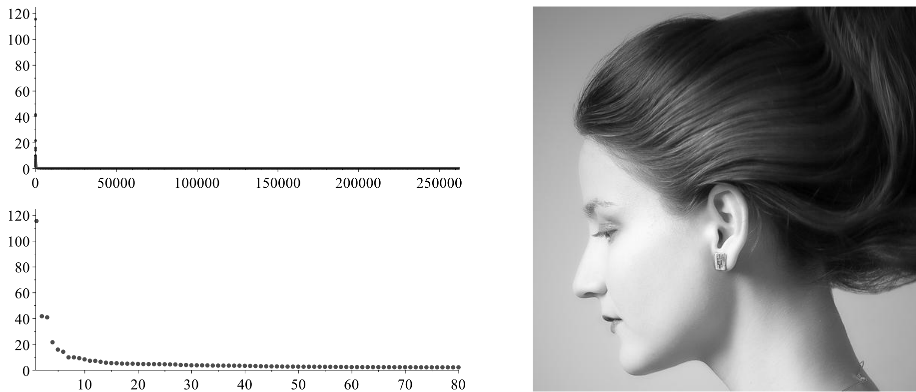


Figure 15.1: On the right we see a 512×512 grayscale image. The graphs on the left show the absolute values of the image's Haar wavelet coefficients, sorted in decreasing order in size. The upper graph shows all of the $512 \times 512 = 262{,}144$ coefficients, while the lower graph zooms into the first 80. There is hardly anything visible in the upper graph because the points mostly coincide with the coordinate axes. In the zoomed lower graph we see the largest coefficient, which is approximately 115, as an isolated dot at the top of the vertical axis. The next few coefficients are approximately 42, 41, 21, 16, and 14. *Photograph courtesy of Stockvault.*

It is this chapter's objective to investigate that question. Experimenting with close-up images of human faces, we find out that retaining around 5 percent of the largest wavelet coefficients suffices to recreate an image which is hardly distinguishable from the original. This observation has practical implications. For one thing, we may save a good deal of storage space by storing a small fraction of the wavelet coefficients instead of a full image. More importantly, if we are willing to accept a slightly deteriorated image, we may save substantially in network transmission time and broadcast bandwidth if we send only the significant wavelet coefficients. All that is needed is a Haar wavelet transformation and truncation at the sending end and an inverse Haar wavelet transformation at the receiving end. Most videos that you watch on the Internet are brought to you this way.

This chapter's program reads a PGM or PPM image, applies the Haar wavelet transform to it, and then discards as many of the smallest wavelet coefficients as possible, while keeping the truncation error below a prescribed threshold. It then reconstructs the image from the remaining wavelets and writes it out to a file. Since the wavelet techniques that we developed in Chapter 13 work with matrices whose dimensions are powers of 2, this chapter's program handles images whose dimensions are powers of 2. Images of arbitrary sizes may be cropped or padded to make them conform to this requirement.

Remark 15.1. The word "discard" can be somewhat ambiguous in this context, so let me clarify. By "discarding a coefficient" I mean setting it to zero. Afterward, when the image is reconstructed as a sum of the products of the coefficients and the corresponding wavelets, a zeroed coefficient does not contribute to the sum; therefore in effect it is "discarded".

Remark 15.2. Applying the wavelet transform to an image's matrix and discarding the small entries leaves us with a *sparse matrix*, that is, a matrix whose vast majority of

entries are zero. The sparse matrix storage techniques of Chapter 11 may be applied quite profitably here to store such transformed and truncated images, but I will not pursue that issue here. You may find it instructive to explore the idea on your own.

15.2 ▪ The truncation error in a grayscale image

We saw in Chapter 13 that the wavelet bases are orthonormal sets; therefore, if $a = (a_{ij})$ are the entries of a grayscale image's wavelet coefficients, then $\|a\| = \big(\sum_{i,j} a_{ij}^2\big)^{1/2}$ is a norm on the space of grayscale images. The *absolute error* after dropping all coefficients below a threshold t is $e_{\text{abs}}(t) = \big(\sum_{|a_{ij}|\le t} a_{ij}^2\big)^{1/2}$. The numerical value of the absolute error by itself, however, is not of much interest. The *relative error* $e_{\text{rel}}(t) = e_{\text{abs}}(t)/\|a\|$ is much more illuminating since it puts the size of the truncation error in context. The relative error is a number between 0 and 1.

Our program receives a grayscale image and a *user-specified relative error* $\hat{e}_{\text{rel}}$ in the interval $[0,1]$. It proceeds to determine the cut-off threshold $\hat{t}$ so that $e_{\text{rel}}(\hat{t}) = \hat{e}_{\text{rel}}$, or equivalently,

$$e_{\text{abs}}(\hat{t}) = \|a\|\hat{e}_{\text{rel}}.$$

The most obvious way of finding $\hat{t}$ is through the *bisection algorithm* (or *bisection method*; see, e.g., [4, 33]). Here is how:

1. Sweep through the matrix a to find the largest and smallest values of $|a_{ij}|$. Call them max and min. While you are at it, also compute $\|a\|$.
2. The cut-off threshold $\hat{t}$ lies somewhere in the interval $[\min, \max]$. As an initial guess, pick t at the midpoint, that is, $t = (\max + \min)/2$.
3. Evaluate the truncation error $e_{\text{abs}}(t) = \big(\sum_{|a_{ij}|\le t} a_{ij}^2\big)^{1/2}$ corresponding to that cut-off value.
4. There are three cases:
   (a) Case $e_{\text{abs}}(t) = \|a\|\hat{e}_{\text{rel}}$: We have found just the right t. Stop.
   (b) Case $e_{\text{abs}}(t) > \|a\|\hat{e}_{\text{rel}}$: We are truncating too much. Set $\max = t$, and go to step 2.
   (c) Case $e_{\text{abs}}(t) < \|a\|\hat{e}_{\text{rel}}$: We are truncating too little. Set $\min = t$, and go to step 2.

In each iteration step, the length of the search interval is cut in half; therefore the iteration will converge in a few steps. Or will it?

Unfortunately, chances are excellent that the iteration described above never will terminate. The problem lies in the stopping criterion in step 4(a). Actually there are two problems with that step. First, a comparison of floating point numbers for equality is inherently problematic since most numbers have no exact representation as floating numbers. A minuscule floating point rounding here and there may throw off the equality. Second, *even in the absence* of floating point rounding, we don't expect the equality $e_{\text{abs}}(t) = \|a\|\hat{e}_{\text{rel}}$ ever to be satisfied in general. The $e_{\text{abs}}(t)$ is computed as a *discrete sum* of terms. Adding or taking away an extra term is likely to move the sum from just below $\|a\|\hat{e}_{\text{rel}}$ to just above it. There may be no combination of a_{ij} terms that hits the target exactly.

The way to fix things is to change the stopping criterion in step 4(a) to something with more certainty. We know that the length of the search interval is halved in every

step, so let's stop if that length is smaller than a small fraction of the image's norm, as in $\max - \min < \epsilon ||a||$, and set ϵ arbitrarily to something small, like 10^{-6}. In summary, we change the stopping condition in step 4(a) to

4(a)': Stop if $\max - \min < \epsilon ||a||$.

Remark 15.3. Although the definitions of the norm and the absolute error involve square roots, there is no need for computing square roots at all in the algorithm described above. For instance, the condition $e_{\text{abs}}(t) > ||a|| \hat{e}_{\text{rel}}$ is equivalent to $e_{\text{abs}}(t)^2 > ||a||^2 \hat{e}_{\text{rel}}^2$, which involves no square roots.

15.3 ▪ The truncation error in a color image

In Chapter 14 we encoded a PPM image as a superposition of three matrices of the image's red, green, and blue intensities. Therefore we may apply the Haar wavelet transform to each of the three matrices, truncate them in conformance to the prescribed relative error $\hat{e}_{\text{rel}}$, and then reconstruct the approximate image from the set of remaining wavelet coefficients. In that respect, color images offer no new challenges. There is a subtle issue to account for, however.

Truncating the three color matrices independently of each other can distort the image's color balance. This is because a pixel's blue component, for instance, may be eliminated while its red component survives. To remain faithful to the original colors, we should change a pixel's RGB colors in unison. Here is what I suggest. Let r_{ij}, g_{ij}, b_{ij} be the entries of the (wavelet transformed) red, green, and blue matrices. Create a composite matrix, let's call it T, such that $T_{ij}^2 = \frac{1}{3}(r_{ij}^2 + g_{ij}^2 + b_{ij}^2)$. That is, T_{ij} is the *root mean square* of r_{ij}, g_{ij}, b_{ij}. Truncate the matrix T in conformance to the prescribed relative error $\hat{e}_{\text{rel}}$, just as if it represented a grayscale image. For any T_{ij} that gets dropped in the process, drop all three of the corresponding r_{ij}, g_{ij}, b_{ij} entries.

15.4 ▪ Image reconstruction

At this point we have discarded an image's wavelet coefficients that fall below a certain threshold. Now we apply the inverse wavelet transform to the truncated coefficients to construct an approximation to the original image.

There is a slight complication here that requires our attention. Let's say that the original image's pixel values (color or grayscale) range from 0 to M. If we apply the wavelet transform to the image, followed by the inverse wavelet transform, without modifying anything in between, the reconstructed image will be identical to the original, and of course the reconstructed pixel values will be integers in the 0 to M range. However, if we modify the transformed matrix, as we do when we discard some of its coefficients, there is no guarantee that the pixel values of the reconstructed image will be integers, nor that they will fall in the 0 to M range. The result would be an illegal PGM or PPM image in general.

In practice, we find that only a handful of the hundreds of thousands of pixel values fall outside the 0 to M range. We treat them by brute force. If a pixel value has fallen below zero, we set it to zero. If it exceeds M, we set it to M. If it is not an integer, we round it to the nearest integer. These are done in the `clip_matrix()` function in subsection 15.6.2.

Listing 15.1: The transcript of an interactive session with this chapter's program. If invoked with too few or improper arguments, it prints a usage message to `stderr` and exits. The image file *profile.pgm* is available at the book's website.

```
$ ./image-analysis
Usage: ./image-analysis rel_err infile outfile
     0.0 ≤ rel_err ≤ 1.0
  1. Reads a PGM or PPM image from infile
  2. Applies the Haar wavelet transform
  3. Sets as many of the smallest wavelet coefficients to zero
     as possible while keeping the relative error
     (in the L^2 norm) just under rel_err
  4. Reconstructs image with the truncated coefficients and
     writes to outfile

$ ./image-analysis 0.05 profile.pgm profile-05.pgm
zeroed 258548 of 262144 wavelet coefficients, 3596 remaining
```

15.5 ▪ The program

This chapter's program relies on the *image-io* module (Chapter 14) for reading and writing images, the *wavelet* module (Chapter 13) for the Haar wavelet transform, the *xmalloc* module (Chapter 7) for memory allocation, and the *array.h* header file (Chapter 8) for making vectors and matrices. Therefore, following the suggestions in Chapters 2 and 6, this project's directory will look like this:

```
$ cd image-analysis
$ ls -F
Makefile  image-analysis.c  image-io.h@  wavelet.h@  xmalloc.h@
array.h@  image-io.c@       wavelet.c@   xmalloc.c@
```

The program is contained entirely in the file *image-analysis.c*, to be detailed below, and compiles into an executable named *image-analysis*. It is expected to be invoked as in

```
$ ./image-analysis 0.12 infile outfile
```

The `infile` is the name of an existing PGM or PPM file; the `0.12` says that we want to truncate the image's wavelet coefficients with a relative error of 0.12, that is, 12 percent. The program does that, constructs an image with the truncated coefficients, and writes it into the file `outfile`. The file names can be anything, but beware that the output will overwrite a file with the same name as the `outfile`, if one exists. The relative error can be anything from 0.0 to 1.0.

Listing 15.1 shows the transcript of an interactive session. If the program is invoked with too few or improper arguments, it prints a usage message to `stderr` and exits. Otherwise it does what is expected of it and prints out a brief statistics regarding its operation.

15.6 ▪ The implementation of *image-analysis.c*

Listing 15.2 gives an outline of the file *image-analysis.c*. I have left out the headers to let you figure out what to include. I will describe the details of the rest of the outline in the following subsections.

Listing 15.2: An outline of the file *image-analysis.c*.

```
1  ▶#include ...
2  ▶static int prune_matrix(double **a, int m, int n, double rel_err) ...
3  ▶static void clip_matrix(double **a, int m, int n, int M) ...
4  ▶static void reduce_pgm_image(struct image *img, double rel_err) ...
5  ▶static void reduce_ppm_image(struct image *img, double rel_err) ...
6  ▶void show_usage(char *progname) ...
7  ▶int main(int argc, char **argv) ...
```

15.6.1 ▪ The function `prune_matrix()`

The function `prune_matrix()` that appears on line 2 of Listing 15.2 receives a pointer `a` to an $m \times n$ matrix and a relative error, `0.0` $\leq$ `rel_err` $\leq$ `1.0`, which is what was called $\hat{e}_{\text{rel}}$ in Section 15.2. It implements that section's bisection algorithm to find the cut-off threshold corresponding to $\hat{e}_{\text{rel}}$. It then zeroes all of the matrix's entries that fall below the threshold and returns the count of the zeroed entries to the caller.

I will describe the workings of the function `prune_matrix()` in words and leave it to you to convert it to C code. You should read and understand Section 15.2 in order to be able to follow these instructions.

1. Find the values maximum, `max`, and minimum, `min`, of the absolute values of `a[i][j]`. Also compute `norm2`, which is square of the norm, that is, $||a||^2$.
   Hint: Let `min = max = fabs(a[0][0]);` and `norm2 = 0.0;`. Scan the matrix's entries in a doubly nested **`for`**-loop, and adjust `max`, `min`, and `norm2` as you go along.

   Important: C has at least two functions for computing absolute values. The function `abs()` computes the absolute value of a number of *integer type*. The function `fabs()` computes the absolute value of a number of *floating point type*. Carelessly confusing the two can lead to nasty and hard-to-spot bugs in your code. Be on guard when using `abs()` and `fabs()`!

   Note: The function `abs()` is declared in *stdlib.h*, while `fabs()` is declared in *math.h*. Do `#include <stdlib.h>` to use `abs()` and `#include <math.h>` to use `fabs()`. In particular, in the case of `fabs()` you may have to link explicitly with the standard mathematics library through the "`-lm`" flag, as is the case with `gcc` on Linux. If you are working on a Unix or Unix-like system, information such as this is at your fingertips in your system's *man pages*. Type, for instance

   ```
   $ man 3 fabs
   ```

   to read all about `fabs()`. The "3" specifies Section 3 of the man pages. Section 3 is the repository of the system's documents on programming libraries. Type

   ```
   $ man man
   ```

   to learn about the organization of the man pages.
2. We wish to find t so that $e_{\text{abs}}(t) = ||a||\hat{e}_{\text{rel}}$, that is, $e_{\text{abs}}(t)^2 = ||a||^2\hat{e}^2_{\text{rel}}$. We have computed $||a||^2$ and are given $\hat{e}_{\text{rel}}$; therefore we may calculate $||a||^2\hat{e}^2_{\text{rel}}$. Call it `abs_err2`. You will need it in the next step.
3. Everything is all set to implement Section 15.2's bisection algorithm for the cut-off threshold t. Do it.

4. Now that we have t, scan the matrix and, wherever $|a_{ij}| < t$, change it to zero. Keep a count of the number of zeroed entries.
5. Return the number of zeroed entries to the caller.

15.6.2 ▪ The function `clip_matrix()`

The function `clip_matrix()` that appears on line 3 of Listing 15.2 receives a pointer `a` to an $m \times n$ matrix which represents (approximately) the pixel values of a grayscale image or one of the red, green, and blue pixel values of a color image. The matrix that arrives here has been constructed through applying the inverse wavelet transform and may not exactly conform to the requirements of a PGM or PPM image. Specifically, its entries are not necessarily integers, and their values are not necessarily in the 0 to M range as they are supposed to be if they were properly formed. The `clip_matrix()` function implements Section 15.4's recommendations for bringing the matrix `a` into conformance with the requirements of *Netpbm*. I will describe what it does in words and leave it to you to convert it to C code. You should read and understand Section 15.4 in order to be able to follow these instructions.

The function `clip_matrix()` scans the matrix `a`:

1. if $a_{ij} < 0$, it sets a_{ij} to zero;
2. else if $a_{ij} > M$, it sets a_{ij} to M;
3. else, it rounds a_{ij} to the nearest integer.

This should be straightforward except possibly for the "round to the nearest integer" part. Since this issue comes up frequently in programming, I will explain the solution as a stand-alone idea so that other chapters may refer to it.

Rounding a floating point number to an integer. Casting a floating point number to an integer type in C discards the number's fractional part. For instance, `(int)3.1` yields 3 and `(int)(-3.1)` yields `-3`. We see that the casting mechanism rounds floating point numbers "toward zero".

This property may be adapted to achieve rounding "toward the nearest integer" through the very common "`0.5 trick`" as follows. Suppose `x = 3.1` and `y = 3.9`. We see that

```
(int)(x + 0.5) = (int)3.6 = 3          // 3.1 is rounded to 3
(int)(y + 0.5) = (int)4.4 = 4          // 3.9 is rounded to 4
```

To round negative numbers "toward the nearest integer", we subtract `0.5`. Thus, if `x = -3.1` and `y = -3.9`, we see that

```
(int)(x - 0.5) = (int)(-3.6) = -3      // –3.1 is rounded to –3
(int)(y - 0.5) = (int)(-4.4) = -4      // –3.9 is rounded to –4
```

Remark 15.4. As an alternative to this "homemade" rounding trick, you may consider the C standard library's `lrint()` function, introduced in C99, which works equally well with negative and positive numbers. Thus, `lrint(3.1)` yields 3 and `lrint(-3.9)` yields `-4`. I continue using the "0.5 trick" out of a force of habit.

Listing 15.3: The function `reduce_pgm_image()`.

```
1  static void reduce_pgm_image(struct image *img, double rel_err)
2  {
3      int m = img->pam.height;
4      int n = img->pam.width;
5      int M = img->pam.maxval;
6      int zero_count;
7      haar_transform_matrix(img->g, m, n, WT_FWD);
8      zero_count = prune_matrix(img->g, m, n, rel_err);
9      haar_transform_matrix(img->g, m, n, WT_REV);
10     clip_matrix(img->g, m, n, M);
11     fprintf(stderr, "zeroed %d of %d wavelet coefficients, %d remaining\n",
12             zero_count, m*n, m*n - zero_count);
13 }
```

15.6.3 ▪ The function `reduce_pgm_image()`

The function `reduce_pgm_image()` that appears on line 4 of Listing 15.2, receives a pointer `img` to a "**struct** image" and a `rel_err` number in the range $[0, 1]$. It expects that `img` points to a PGM image. (It doesn't check for this!) Here is what it does:

1. applies `haar_transform_matrix()` (with the `WT_FWD` flag) to the image's matrix;
2. applies `prune_matrix()` to zero those matrix entries that fall below the cut-off threshold corresponding to the prescribed relative error `rel_err`;
3. applies `haar_transform_matrix()` (with the `WT_REV` flag) to reconstruct the image from the truncated coefficients;
4. applies `clip_matrix()` to bring the matrix's entries into conformance with the PGM's requirements; and
5. prints a brief report on the number of coefficients that were zeroed.

Listing 15.3 shows my implementation of the function.

15.6.4 ▪ The function `reduce_ppm_image()`

The function `reduce_ppm_image()` that appears on line 5 of Listing 15.2 receives a pointer `img` to a "**struct** image" and a `rel_err` number in the range $[0, 1]$. It expects that `img` points to a PPM image. (It doesn't check for this!) It performs exactly the same tasks as the `reduce_pgm_image()` function described above, so I won't repeat them. However, you will need to read Section 15.3 carefully regarding the construction of the T matrix because that's what will be sent to `prune_matrix()` for truncation.

15.6.5 ▪ The function `show_usage()`

The function `show_usage()` that appears on line 6 of Listing 15.2 prints a brief help about how to invoke the program. You can see the output of my implementation's `show_usage()` in Listing 15.1. It is more detailed than is customary in such things, but I was driven to it because despite having worked with this program for many years, I tend to forget the meanings of its command-line arguments and what the program is meant to do. The `show_usage()` prints a little handy reminder.

Listing 15.4: The function `main()` in its entirety. The midblock declarations on lines 20 and 21 are allowed in C99 but not in C89. If you wish to reduce the code to C89, then follow the instructions regarding **Line 12** on page 130.

```
1  int main(int argc, char **argv)
2  {
3      char *infile, *outfile, *endptr;
4      double rel_err;
5
6      if (argc ≠ 4) {
7          show_usage(argv[0]);
8          return EXIT_FAILURE;
9      }
10     rel_err = strtod(argv[1], &endptr);
11     if (*endptr ≠ '\0' || rel_err < 0.0 || rel_err > 1.0) {
12         fprintf(stderr, "*** the rel_err argument should be "
13                 "between 0.0 and 1.0\n");
14         return EXIT_FAILURE;
15     }
16     infile = argv[2];
17     outfile = argv[3];
18
19     pm_init(argv[0], 0);
20     struct image *img = read_image(infile);
21     struct pam *pam = &img→pam;
22     if (pam→format == PGM_FORMAT || pam→format == RPGM_FORMAT)
23         reduce_pgm_image(img, rel_err);
24     else if (pam→format == PPM_FORMAT || pam→format == RPPM_FORMAT)
25         reduce_ppm_image(img, rel_err);
26     else {
27         fprintf(stderr, "*** file %s, line %d: shouldn't be here\n",
28                 __FILE__, __LINE__);
29         return EXIT_FAILURE;
30     }
31
32     write_image(outfile, img);
33     free_image(img);
34     return EXIT_SUCCESS;
35 }
```

15.6.6 ▪ The function `main()`

The function `main()` that appears on line 7 of Listing 15.2 is the driver of the other functions that have been discussed up to this point. It parses the command-line for the user-specified relative error and the input and output file names. It reads the image from the input file into a "**`struct`** `img`", detects the type of the image (PGM or PPM), and then dispatches the images to `reduce_pgm_image()` or `reduce_ppm_image()` for processing. Subsequently, it writes the modified image into the output file and exits. Listing 15.4 shows my implementation of `main()`. I will proceed to describe some of its more subtle points.

Line 6. We expect the program to be invoked with three arguments (see Section 15.5); therefore `argc` should be 4.

Lines 10–17. The user-specified relative error is in `argv[1]` as a string. Here we apply the standard library's `strtod()` function (see Chapter 5) to extract its numerical value.

Line 19. This initializes the *libnetpbm* library. See Section 14.5 for explanation.

Line 20. We call `read_image()` (see subsection 14.8.1) to read the image file that's given on the command-line.

Line 21. There are frequent references to `img→pam` in the rest of the code. We define a shorthand here to reduce clutter. (We did the same in many place in Chapter 14.)

Lines 22–30. The meanings of preprocessor symbols `PGM_FORMAT`, etc., are described in Section 14.5. Here we test for the image type. If it is a plain PGM or raw PGM, we invoke `reduce_pgm_image()`. If it is a plain PPM or raw PPM, we invoke `reduce_ppm_image()`. Otherwise we print an error message and exit because we are not equipped to handle other types of images.

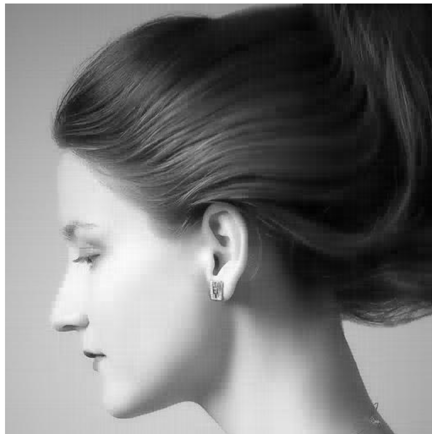
On line 27 I say "shouldn't be here" and I mean it. The function `read_image()` (see subsection 14.8.1) checks for the image type. If anything other than a PGM or PPM, it prints an error message and exits the program. Therefore, if the program is functioning correctly, line 27 cannot be reached. If it is reached, then there is a bug.

Lines 32–34. We are done with image processing. We call `write_image()` (see subsection 14.8.3) to write the reconstructed image to the specified output file, free the allocated memory, and exit.

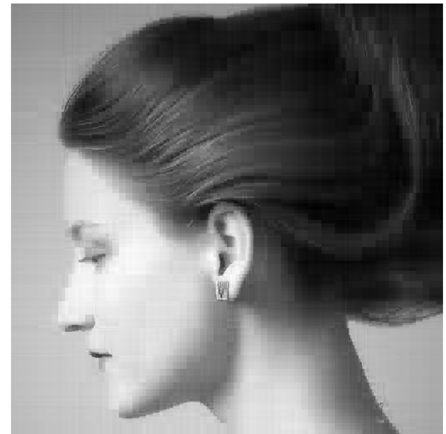
15.7 ▪ *Project Image Analysis*

Complete the file *image-analysis.c*, compile, and test. You will find several PGM and PPM images in the book's website for your tests. For your reference, Figure 15.2 shows the results I obtained on the 512×512 PGM image, *profile.pgm*. The unaltered image is shown in Figure 15.1 on page 136.

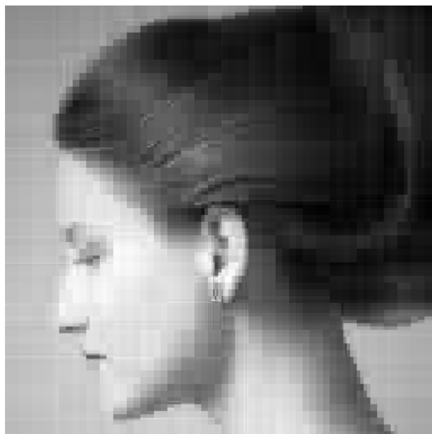
Observe that the image obtained by setting $\hat{e}_{\text{rel}} = 0.05$ has suffered only a minor deterioration relative to the original. Also observe that it is constructed as a sum of only $3,596$ wavelets, while the original image is a sum of $512 \times 512 = 262,144$ wavelets. In a sense, the wavelet analysis of the image has achieved a compression ratio of $262,144/3,596 \approx 73$.



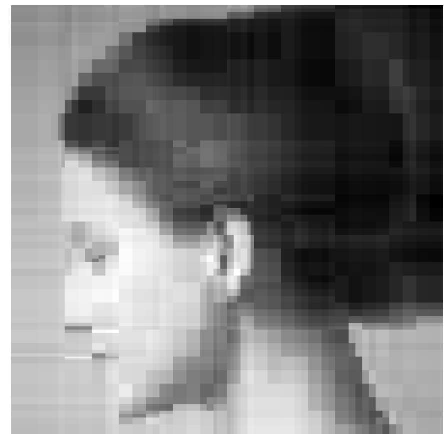
$\hat{e}_{\text{rel}} = 0.03$, nonzero coeffs $= 13,163$



$\hat{e}_{\text{rel}} = 0.05$, nonzero coeffs $= 3,596$



$\hat{e}_{\text{rel}} = 0.07$, nonzero coeffs $= 1,163$



$\hat{e}_{\text{rel}} = 0.09$, nonzero coeffs $= 489$

Figure 15.2: Results of four experiments with the image *profile.pgm*. The unaltered image is shown in Figure 15.1. *Original image courtesy of Stockvault.*

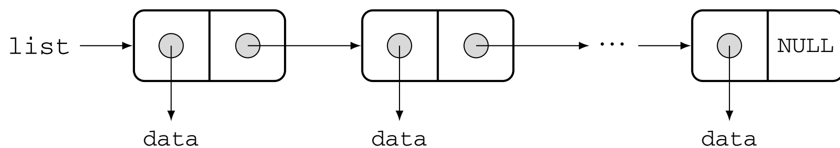
Chapter 16

Linked lists

Prerequisite: *Chapter 7*

16.1 ▪ Linked lists

A *linked list* is a data structure consisting of objects called *nodes*, where each node consists of a pointer to user-supplied data and a pointer to the next node, as shown in this schematic diagram:



The last node has no successor; therefore its next-node pointer is NULL. Access to the linked list is provided through a pointer to the first node—it's marked list in the diagram above, but you may name it anything you wish. Beginning with list, we may hop from a node to the next following the next-node pointers and thus traverse the entire list. When we arrive at NULL, we know we're at the end. The C idiom for traversing a linked list is

```
for (p = list; p ≠ NULL; p = p→next {
    ... do whatever with p ...
}
```

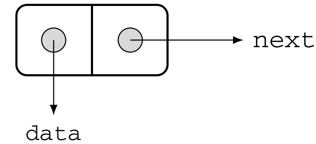
where p has the type of a "pointer to node".

Linked lists are particularly useful when dealing with collections of variable size. For instance, in the evolution model of Chapter 17 we keep track of the "animals" in the simulated world by chaining them into a linked list. When an animal dies, we remove the corresponding node from the list. When an animal is born, we insert a node in the list. Thus, the linked list shrinks and expands as the animal population changes.

C has no standardized linked lists facility. Some other languages do. The containers library in C++, for instance, provides extensive support for linked lists. The *Lisp* programming language and its derivatives have linked lists as their primary data types. (The name "*Lisp*" is derived from "List processing".) This chapter's algorithms for manipulating linked lists are influenced by *Lisp* and *Lisp* idioms.

In the *Lisp* jargon, a node is called a *cons cell*[48], and that's what we will call it here to carry on the tradition. A cons cell is very simple object; it is a pair of pointers, one of which points to data and the other points to another cons cell. We introduce a new *type*, a `conscell`, using C's **`typedef`** mechanism:

```
typedef struct conscell {
    void *data;
    struct conscell *next;
} conscell;
```



The structure's `data` member is a void pointer which is—as all void pointers are—capable of pointing to objects of *any* type; see the discussion of void pointers in Section 4.3. The structure's `next` member points to a "**`struct`** `conscell`" and thus enables the chaining of cons cells.

16.2 ▪ The program

The goal of this chapter is to implement a *linked lists* module which is made up of an interface file, *linked-list.h*, shown in its entirety in Listing 16.1, and an implementation file, *linked-list.c*, which I will present in fragments in the rest of this chapter. It is your job to assemble the pieces and complete *linked-list.c*.

The module consists of a suite of functions for manipulating linked lists. The purpose and implementation of each function is explained in the individual sections below. Additionally, each section offers a suggested driver (files *test[1–7].c*) to exercise that function and presents a transcript of a sample interactive session to serve you as a guide. The *Xmalloc* module of Chapter 7 is used throughout for allocating memory. Therefore, following the suggestions of Chapters 2 and 6, this project's directory will look like this:

```
$ cd linked-lists
$ ls -F
Makefile       test1.c  test4.c  test7.c  xmalloc.c@
linked-list.c  test2.c  test5.c  test8.c  xmalloc.h@
linked-list.h  test3.c  test6.c  test9.c
```

All function names in this module begin with a leading `ll_` as a way of identifying them with the *linked lists* module. This is a customary way of naming a suite of related functions in C.

16.3 ▪ The function `ll_push()`

The function `ll_push()` that appears on line 7 of Listing 16.1 is the simplest and most basic of this chapter's suite of list manipulation functions. Its purpose is to prepend a node, that is, a cons cell, to a linked list. Specifically, it receives a pointer to a list and a pointer to some data. It creates a new cons cell and makes its `data` member point to the given data and its `next` member point to the list's head. It returns the address of the cons cell, which is now the head of the linked list. Listing 16.2 shows the function's self-evident implementation. Study it closely, and then insert it in your *linked-list.c*.

---
[48]That name is derived from *construct* or *constructor*.

Listing 16.1: The header file *linked-list.h* provides the interface of the *linked lists* module.

```
1  #ifndef H_LINKED_LISTS_H
2  #define H_LINKED_LISTS_H
3  typedef struct conscell {
4      void *data;
5      struct conscell *next;
6  } conscell;
7  conscell *ll_push(conscell *list, void *data);
8  conscell *ll_pop(conscell *list);
9  void      ll_free(conscell *list);
10 conscell *ll_reverse(conscell *list);
11 conscell *ll_sort(conscell *list,
12             int (*cmp)(const void *a, const void *b, void *params),
13             void *params);
14 conscell *ll_filter(conscell *list,
15             int (*filter)(const void *a),
16             conscell **removed);
17 int      ll_length(conscell *list);
18 #endif /* H_LINKED_LISTS_H */
```

Listing 16.2: The function `ll_push()` in *linked-list.c* prepends a node to a linked list.

```
1  conscell *ll_push(conscell *list, void *data)
2  {
3      conscell *new = xmalloc(sizeof *new);
4      new→data = data;
5      new→next = list;
6      return new;
7  }
```

16.3.1 ▪ A demo of `ll_push()`: *test1.c*

The file *test1.c* in Listing 16.3 provides a driver demonstrating a typical use of `ll_push()`. It begins with an empty linked list, then pushes four items into it, and then scans the list and prints the data values stored in it. Let's go through the code line by line:

Line 5. The `list` variable declared here points to the first element of our linked list. It is initialized to `NULL`, indicating that the list is empty, i.e., it has no nodes.

Line 6. A few arbitrary numbers are introduced here to serve as data in our demonstration.

Line 7. We pass the address of the first number, `&a`, to `ll_push()`, which creates a cons cell and prepends it to the list and returns the address of the updated list. We update the value of `list` by assigning the new address to it. The next three lines repeat the process with `&b`, `&c`, and `&d`.

Line 12. We traverse the list and print the data stored in it. Since the data pointers are void pointers, they cannot be dereferenced as they are; that's the reason for casting `p→data` to `(int *)` before dereferencing.

Listing 16.3: File *test1.c* provides a demo of `ll_push()`. It begins with an empty linked list and then pushes four items into it.

```
1  #include <stdio.h>
2  #include "linked-list.h"
3  int main(void)
4  {
5      conscell *list = NULL;        // an empty list
6      int a = 101, b = -45, c = 1000, d = 12;
7      list = ll_push(list, &a);
8      list = ll_push(list, &b);
9      list = ll_push(list, &c);
10     list = ll_push(list, &d);
11     printf("the original linked list:\n");
12     for (conscell *p = list; p ≠ NULL; p = p→next)
13         printf("%d ", *(int *)p→data);
14     putchar('\n');
15     return 0;
16 }
```

We compile the program into an executable called *test1* and run it. We get

```
$ ./test1
the original linked list:
12 1000 -45 101
```

Note the reversed order relative to the original `a`, `b`, `c`, and `d`. Can you explain why that happens?

Remark 16.1. Don't take the program in *test1.c* as a finished product. A well-behaved program will free all allocated memory before it exits. This one doesn't. You may want to return here and fix *test1.c* after you read about `ll_free()` in Section 16.5.

16.4 ▪ The function `ll_pop()`

The function `ll_pop()` that appears on line 8 of Listing 16.1 is almost an inverse of `ll_push()`—it removes a linked list's head node. Specifically, it frees the memory allocated previously for that node's cons cell and returns the address of the list's new head. We design `ll_pop()` so that it is safe to call it with an empty list; since there are no nodes to remove, it will do nothing. Generally, the function is used as

```
list = ll_pop(list);
```

After popping a node, the associated data becomes inaccessible. Grab and save the data if you are going to need it. If you have explicitly allocated memory for the data, it is your responsibility to free that memory before you pop the node. The function `ll_pop()` itself has no knowledge of the nature of data that the cons cell points to.

Write an implementation of `ll_pop()`—it's not hard—and add it to your *linked-list.c*.

16.4.1 ▪ A demo of `ll_pop()`: *test2.c*

Copy *test1.c* to *test2.c*, and add code to pop two nodes then print the result. Then pop three more nodes and print the result. Here's what mine says:

```
$ ./test2
the original linked list:
12 1000 -45 101
the list after popping two nodes:
-45 101
the list after popping three more nodes:
```

The last step prints nothing because the list is empty. Note that applying `ll_pop()` five times to a linked list of four nodes is permissible according to our specifications of `ll_pop()`.

16.5 ▪ The function ll_free()

The function `ll_free()` that appears on line 9 of Listing 16.1 frees the memory allocated for *all the cons cells* of a linked list. Normally you will apply `ll_free()` to a linked list when you are done with it.

Have in mind that `ll_free()` has no knowledge of the nature of the data that the list's nodes point to. If you have explicitly allocated memory for the data, you will have to traverse the list and free that memory before calling `ll_free()`.

Here is a possible implementation of `ll_free()`:

```
void ll_free(conscell *list)                // version 1: iterative
{
    while (list ≠ NULL) {
        conscell *p = list→next;            // remember the next node
        free(list);                         // free the head node
        list = p;                           // produce a shorter list
    }
}
```

Note the delicate "tap dance" there. If we free a node prematurely, we will lose its next-node pointer and therefore will be cut off from the rest of the list. The trick is to look one node ahead and save its address before freeing the current node.

The implementation above is an *iterative* algorithm—it walks through the linked list in a **while**-loop and performs an operation on each node until it reaches the end. Alternatively, `ll_free()` may be implemented in a *recursive* algorithm that takes quite a different point of view. I will describe the very *Lisp*-ish algorithm in words first:

1. If the list is empty, then there is nothing to do.
2. Otherwise detach and free the head node and apply `ll_free()` to the rest of the list.

Here is the C implementation of that algorithm. Note that in contrast to the iterative version, this one has no explicit looping constructs at all!

```
void ll_free(conscell *list)                // version 2: recursive
{
    if (list ≠ NULL) {
        conscell *p = list→next;            // remember the next node
        free(list);                         // free the head node
        ll_free(p);                         // recurse: free the rest of the list
    }
}
```

Choose one or the other implementation of `ll_free()`, and insert it in your *linked-list.c*. They have equivalent effects, so the choice does not matter for our purposes, but it would be an excellent idea if you tried both methods for the sake of experimentation.

16.5.1 ▪ A demo of `ll_free()`: *test3.c*

Copy *test1.c* to *test3.c*. Add code to free the linked list before exiting, and print what it does, as in

```
ll_free(list);
printf("linked list memory freed\n");
```

Here is what mine says:

```
$ ./test3
the original linked list:
12 1000 -45 101
linked list memory freed
```

Remark 16.2. The `ll_free()` function frees the memory allocated for a linked list's cons cells. It *does nothing* to free the memory allocated, if any, for the corresponding data. It is the programmer's responsibility to free all memory associated with the data before calling `ll_free()`. To make this chapter's examples as simple as possible, I have limited the sample data to numerical values that require no memory allocation and therefore no freeing. In the more practical instances that we will encounter in the future chapters, memory for data is always allocated and must be freed.

Remark 16.3. After `ll_free(list)` returns, the value of `list` is meaningless; it points to an area of memory that no longer belongs to you, and of course you should not even consider referring to it. Generally this is not a problem—why would you want to refer to a nonexistent list, anyway?—but some programmers derive comfort from setting the value of a freed pointer to `NULL`. If that's what you want, then you will have to set `list = NULL` explicitly after `ll_free(list)` returns.

Alternatively, you may redesign `ll_free()` completely so that instead of `list`, it takes the `list`'s *address*, as in `ll_free(&list)`, for argument. It this way it will know where `list` is stored, and therefore it will be able to set it to `NULL` before it returns, thus freeing the caller from the concern of `NULL`-ing the pointer explicitly. Hanson [26], for instance, takes this approach.

16.6 ▪ The function `ll_reverse()`

You have observed that when we assemble a linked list one node at a time, the list comes out in the reverse order of the insertions. This is not of concern in many cases, but in a few cases where it is, we should like to be able to reverse the order of the nodes of a given list. The function `ll_reverse()` that appears on line 10 of Listing 16.1 on page 149 does exactly that. It receives a list and returns the address of the reversed list:

```
list = ll_reverse(list);
```

An algorithm for `ll_reverse()` may not be immediately obvious—I know, it wasn't obvious to me the first time I thought about it—but once you see it, it turns out to be absolutely trivial. I urge you, therefore, to *stop reading now* and spend a few minutes thinking about how *you* would go about reversing a linked list. After you figure it out (or if you give up) come back and read on.

The idea is this: think of two lists, $list_1$ which is the original one, and $list_2$ which is initially empty. Remove the first node from $list_1$, and prepend it to $list_2$. Repeat until $list_1$ is exhausted. Then $list_2$ is the desired reversed list. Return $list_2$ to the caller. Done.

Write an implementation of the `ll_reverse()`, and add it to your collection in *linked-list.c*.

16.6.1 ▪ A demo of `ll_reverse()`: *test4.c*

To test your work, copy *test3.c* to *test4.c*. Reverse the original list, and print it out. Here is what mine says:

```
$ ./test4
the original linked list:
12 1000 -45 101
the reversed linked list:
101 -45 1000 12
linked list memory freed
```

16.7 ▪ The function `ll_sort()`

The purpose of the function `ll_sort()` that appears on line 11 of Listing 16.1 on page 149 is to reorder a list's nodes according to a prescribed comparison criterion. The declaration of `ll_sort()` in that listing is somewhat long; therefore I have broken it into three lines. This should not obscure the fact that `ll_sort()` takes three arguments:

1. The first argument, `list`, points to the linked list to be sorted.
2. The second argument, `cmp()`, is a user-supplied *comparison function* which conforms to the prototype

   ```
   int cmp(const void *a, const void *b, void *params);
   ```

   and which is the subject of subsection 16.7.1.
3. The third argument, `params`, is a pointer to a set of parameters which the comparison function may or may not need. The function `ll_sort()` passes that pointer to the comparison function whenever it calls it.

The function `ll_sort()` returns a pointer to the head of the reordered list; therefore a typical call looks like this:

```
list = ll_sort(list, cmp, params);
```

If the comparison function has no use for parameters, then passing `NULL` for `params` will do

```
list = ll_sort(list, cmp, NULL);
```

In the next subsection I will elaborate on the nature of the comparison function. In the subsection after that I will describe the sorting algorithm.

16.7.1 ▪ The comparison function

I will explain the specifics of `ll_sort()`'s comparison function through the following two examples.

Example 1: Suppose the data associated with each node is an integer. We wish to sort the list in the increasing (or decreasing) order of those numbers.

Example 2: Suppose the data associated with each node is a coordinate pair (x, y). We wish to sort the list according to increasing distances away from a given point $p = (x_0, y_0)$.

In each case the comparison function takes a pair of data items and decides which one should come first in the list. Customarily, a compassion function `cmp(a,b)` returns an integer less than, greater than, or equal to zero if the first argument is considered to be "less than", "greater than", or "equal" the second, respectively, for sorting purposes. A reasonable comparison function for **Example 1** above would be

```
int cmp1(int a, int b)        // tentative
{
    if (a < b)
        return -1;
    else if (a > b)
        return 1;
    else
        return 0;
}
```

For **Example 2** it would make sense to introduce a structure to hold a coordinate pair

```
struct point { double x; double y; };
```

and then pass the addresses of such structures to the comparison function, as in

```
int cmp2(struct point *a, struct point *b, struct point *p) // tentative
{
    double dx, dy, d1, d2;
    dx = a->x - p->x;          // the x distance of a from p
    dy = a->y - p->y;          // the y distance of a from p
    d1 = dx*dx + dy*dy;        // square of distance of a from p
    dx = b->x - p->x;          // the x distance of b from p
    dy = b->y - p->y;          // the y distance of b from p
    d2 = dx*dx + dy*dy;        // square of distance of b from p
    if (d1 < d2)
        return -1;
    else if (d1 > d2)
        return 1;
    else
        return 0;
}
```

The function calculates the distances (actually the squares of the distances) of the points `a` and `b` from `p` and returns `-1`, `0`, or `+1` depending on the relationship between those distances.

Unfortunately neither of the two preceding comparison functions is suitable for our purposes since their arguments are so tightly coupled with their data types. We want the sorting function `ll_sort()` to be *generic*, that is, applicable to *any* linked list, regardless of the associated data types. With that in mind, we postulate a universal comparison function in the form[49]

```
int cmp(const void *a, const void *b, void *params);
```

[49] The purpose of the **`const`** qualifiers here is to reassure the compiler that the data that `a` and `b` point to will be read but not modified. This can help the compiler produce a better optimized code; otherwise they are not essential. Don't worry about them if they look unfamiliar.

Since a void pointer is compatible with pointers to all data types, this form of the comparison function is ideal; pointers to data of any type may be passed to it. The function's first two arguments point to the items that are to be compared. The third argument points to parameters that may or may not be useful for the purpose of that comparison. Modifying the previous comparison function cmp1() into the new form, we arrive at

```
int cmp1(const void *aa, const void *bb, void *pp)    // correct form
{
    int a = *(int *)aa;
    int b = *(int *)bb;
    if (a < b)
        return -1;
    else if (a > b)
        return 1;
    else
        return 0;
}
```

The function casts the void pointer aa to a pointer-to-**int** and then dereferences it to obtain its value, a. It does the same with bb. Then it proceeds just as in the previous version of cmp1(). The parameter pp is not used. Similarly, the modified version of cmp2() takes the form

```
int cmp2(const void *aa, const void *bb, void *pp)    // correct form
{
    struct point *a = (struct point *)aa;
    struct point *b = (struct point *)bb;
    struct point *p = (struct point *)pp;
    double dx, dy, d1, d2;

    dx = a->x - p->x;
    dy = a->y - p->y;
    d1 = dx*dx + dy*dy;        // square of distance of a from p
    dx = b->x - p->x;
    dy = b->y - p->y;
    d2 = dx*dx + dy*dy;        // square of distance of b from p
    if (d1 < d2)
        return -1;
    else if (d1 > d2)
        return 1;
    else
        return 0;
}
```

Now cmp1() and cmp2() have identical signatures/prototypes, and that's what matters. Their innards are quite different.

16.7.2 ▪ Quicksort

Sorting is a huge subject and there are books written on it. See, e.g., Knuth's *The Art of Computer Programming* [34]. I will describe what is known as the *quicksort* algorithm, which, in its recursive form, is particularly well suited to sorting linked lists. Here is the algorithm in words:

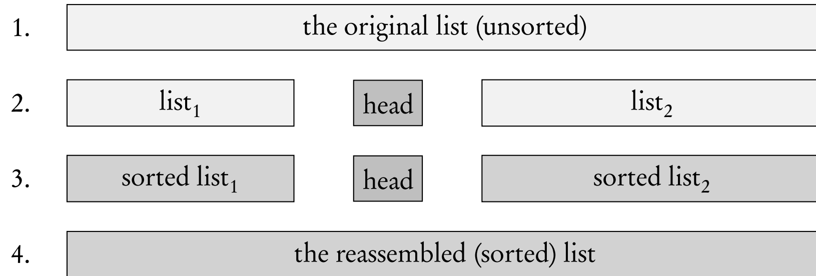


Figure 16.1: This diagram shows the four steps of the quicksort algorithm. Line 1 represent the original (unsorted) list. In line 2 we have isolated the list's head and split the rest into $list_1$ and $list_2$, where all items in $list_1$ are "less than" the head (according to the comparison criterion) and $list_2$ consists of what's left. In line 3 we have sorted $list_1$ and $list_2$, and in line 4 we have pasted the three pieces to form a completely sorted list.

1. If the list is empty, there is nothing to do. The result is the empty list.
2. Otherwise, detach the list's first node (the head) and set it aside. Split the rest into two lists, $list_1$ and $list_2$, where $list_1$ consists of all the nodes that are "less than" the first node (according to the comparison function) and $list_2$ consists of the others.
3. Sort $list_1$ and $list_2$.
4. Reassemble the list as $list_1 \to head \to list_2$. This is the desired sorted list.

The procedure is highly recursive since the sorting of the sublists $list_1$ and $list_2$ in step 3 invokes the sorting procedure all over again. Figure 16.1 shows the procedure in a diagrammatic way. Listing 16.4 gives a quite literal implementation of that algorithm in C. Study Listing 16.4 carefully, and be sure that you understand every bit of it; then add it to your *linked-list.c*.

Remark 16.4. Although the `ll_sort()` function is unabashedly recursive in its outline, the **`while`**-loop embedded in the middle of Listing 16.4 is iterative. Is it possible to replace it with recursion as well? Certainly! The purpose of that **`while`**-loop is to split a given list into $list_1$ and $list_2$ according to a given criterion. The function `ll_filter()`, which is the subject of the next section, does exactly that and is strictly recursive. Therefore one may, in principle, replace the **`while`**-loop in `ll_sort()` with a call to `ll_filter()` and obtain a purely recursive `ll_sort()`. Unfortunately the coding details of passing arguments from `ll_sort()` to `ll_filter()` become somewhat cumbersome and obscure the conceptual simplicity of the underlying idea.[50] You may want to try it for yourself to see that.

16.7.3 ▪ A demo of `ll_sort()`: *test5.c*

To test/demo `ll_sort()`, copy *test3.c* to *test5.c*. Add two comparison function, `cmp1()` and `cmp2()`, one for sorting the linked list in the ascending order of its numerical data and the other in the descending order. Compile and run the program. Here is what mine says:

[50]This is not a shortcoming of the idea. Rather, it is an indication that we have reached the limits of what one may express comfortably in C. A *functional programming language*, such as *Lisp*, on the other hand, can express the idea elegantly and succinctly.

Listing 16.4: A recursive implementation of the quicksort algorithm.

```
1  conscell *ll_sort(conscell *list,
2          int (*cmp)(const void *a, const void *b, void *params),
3          void *params)
4  {
5      conscell *list1 = NULL;
6      conscell *list2 = NULL;
7      conscell *p, *q, *head;
8      if (list == NULL)
9          return list;
10     head = list;
11     p = list→next;
12     while (p ≠ NULL) {
13         q = p→next;                              // remember the next node
14         if (cmp(p→data, head→data, params) < 0) {
15             p→next = list1;
16             list1 = p;
17         } else {
18             p→next = list2;
19             list2 = p;
20         }
21         p = q;
22     }
23     list1 = ll_sort(list1, cmp, params);     // recursion
24     list2 = ll_sort(list2, cmp, params);     // recursion
25     head→next = list2;                        // prepend head to list2
26
27     if (list1 == NULL)
28         return head;
29
30     for (p = list1; p→next ≠ NULL; p = p→next)  // find list1's tail node
31         ;   // empty body!
32
33     p→next = head;
34     return list1;
35 }
```

```
$ ./test5
the original linked list:
12 1000 -45 101
the list sorted in the ascending order:
-45 12 101 1000
the list sorted in the descending order:
1000 101 12 -45
linked list memory freed
```

16.8 ▪ The function `ll_filter()`

Filtering a linked list splits it into two lists: one consisting of the items that are "caught" by a given filter and the other of the items that have "passed through". The user supplies a function, we call it a *filter*, that examines a data item and returns `0` if the item is to be passed through and nonzero if it is to be caught. For instance, the function

```
int catch_the_odds(int a)          // tentative form
{
    return a%2;
}
```

is a filter that catches odd integers and lets the evens pass through. For the reasons explained in the previous section, this function will have to be modified into the generic form

```
int catch_the_odds(void *aa)       // correct form
{
    int a = *(int *)aa;
    return a%2;
}
```

in order to mesh with the `ll_filter()` function which we intend to develop in this section and which appears on line 14 of Listing 16.1 on page 149. The declaration of `ll_filter()` in that listing is long; therefore I have broken it into three lines. The function takes three arguments:

1. The first argument, `list`, points to the linked list that is to be filtered.
2. The second argument, `filter()`, is a user-supplied filter function as described above.
3. The third argument, `removed`, is the address of a pointer to the linked list of the items caught by the filter.

The function returns the linked list of the items that pass through the filter. A typical use of the `filter()` takes the form

```
conscell *list, *removed = NULL;
... create the list, then ...
list = ll_filter(list, filter, &removed);
```

After the last line, `removed` points to the linked list of the caught items, and `list` points to the linked list of the items that have passed through.

As to the implementation of `ll_filter()`, both iterative of recursive strategies are possible. In the iterative strategy we keep track of the address of the *previous* node as we traverse the list. If the current node is to be removed, we make the current node's previous node point to the current node's next node. I will leave it to you to implement this if you wish. It's an instructive exercise; try it, it's not very hard.

The recursive strategy, again very *Lisp*-ish, is short and sweet. Here it is in words:

1. If the list is empty, there is nothing to do; return the empty list.
2. Otherwise, if the filter catches the list's head, (a) detach the head from the list, (b) attach it to the removed list, and (c) filter and return the rest of the list.
3. Otherwise (a) detach the head from the list, (b) filter the rest of the list, (c) reattach the head, and (d) return the list.

This maps essentially word for word to C. Study the following code, and then insert it into your *linked-list.c*:

```
conscell *ll_filter(conscell *list,
        int (*filter)(const void *a), conscell **removed)
{
    if (list == NULL)
```

```
        return list;
    else if (filter(list→data)) {
        conscell *p = list→next;    // remember the next node
        list→next =  *removed;
        *removed = list;
        return ll_filter(p, filter, removed);
    } else {
        list→next = ll_filter(list→next, filter, removed);
        return list;
    }
}
```

16.8.1 ▪ A demo of `ll_filter()`: *test6.c*

To test/demo `ll_filter()`, copy *test3.c* to *test6.c*, define a filter that catches the odd integers, apply the filter to the existing linked list, and print the caught and passed through lists. Here is what mine says:

```
$ ./test6
the original list:
12 1000 -45 101
the list after removing the odds:
12 1000
these items were removed:
101 -45
memory of passed-through list freed
memory of caught list freed
```

16.9 ▪ The function `ll_length()`

The function `ll_length()`, whose prototype is given on line 17 of Listing 16.1 on page 149, returns the length, that is, the number of nodes, of a linked list. As with the other `ll_*()` functions, this may also be implemented in an iterative or recursive fashion. Write one or the other version (or both!), and insert it in your *linked-list.c*.

To test/demo your work, copy *test3.c* to *test7.c* and add code to write and print the number of nodes. Here is what mine says:

```
$ ./test7
the original linked list:
12 1000 -45 101
the list has 4 nodes
linked list memory freed
```

16.10 ▪ *Project Linked Lists*

Part 16.1. Complete and test your *linked-list.[ch]* and the driver files *test[1–7].c* as instructed in the preceding sections.

Part 16.2. [optional] Our `ll_sort()` handles comparison functions which may depend on a parameter (which we named `params`). Modify the function `ll_filter()` to handle filter functions that depend on a parameter in a similar way.

Part 16.3. [optional] Extend the *linked lists* module by adding a function `ll_append()` with the prototype

```
conscell *ll_append(cons struct *list1, cons struct list2);
```

which concatenates `list1` and `list2`; that is, it makes `list1`'s last node point to `list2`'s first node. It returns the address of the new list. The return value is redundant since it is no different from the address of `list1`. Nonetheless, it is a nice touch to do so since that conforms to the style of the other `ll_*()` functions.

Write a driver, *test8.c*, to test/demonstrate the function.

Part 16.4. [optional] Extend the *linked lists* module by adding a function `ll_map()` with the prototype

```
conscell *ll_map(conscell *list, void (*map)(void *data));
```

which applies a given function, `map()`, to the data at every node of a linked list. For instance, if we let

```
void square_me(void *data)
{
    int *n = data;
    *n = (*n) * (*n);
}
```

then calling `list = ll_map(list, square_me)` with *test1.c*'s linked list should produce

```
$ ./test9
the original list:
12 1000 -45 101
the squared list:
144 1000000 2025 10201
linked list memory freed
```

Write a driver, *test9.c*, to demonstrate this.

Chapter 17
The evolution of species

Prerequisites: Chapters 7, 8, 9, 10, 16

17.1 ▪ Introduction

In the late 1980s Michael Palmiter devised a very simple model of evolution by natural selection which exhibited a strikingly realistic behavior.[51] A. K. Dewdney popularized it in a *Scientific American* article [17]. I learned about these from Conrad Barski's book [7], where he develops a variant of the simulation in *Lisp*. In this chapter we will write a C version and introduce variations of our own. The models appear to be quite robust and insensitive to the details of the evolutionary mechanisms in the sense that all variants exhibit similar evolutionary characteristics.

The simulation begins with one or more individuals endowed with random genetic structures whose descendants reproduce, evolve, and gradually adapt to their environments in the course of many (millions of) generations. In the long run, genetically distinct populations emerge in environmentally distinct regions of the simulated world.

Let me first briefly describe a rough overview of the basic ideas; then I will supply the details.

In the simulation, time flows in discrete "time ticks". The "world" consists of a two-dimensional rectangular grid of cells. "Animals" move about in the world, hopping from a cell to an adjacent cell at each time tick, expending a certain amount of energy on each hop. If an animal's energy drops to zero, it dies and is removed from the simulation. Food sprouts in random cells. If an animal stumbles upon a cell containing food, it eats the food and gains energy. If an animal's energy increases past a certain threshold (the reproduction threshold), it splits into two animals in a manner of asexual reproduction, and each of the two inherits half of the parent's energy.

Each animal is endowed with a "chromosome" which is a vector of length 8 of positive integers. The chromosome remains unchanged over an animal's lifetime. The vector's eight entries are the "genes". If one or more of an animal's genes are distinctly greater than the rest, we say that those are its "dominant genes". The chromosome is the only characteristic that distinguishes one animal species from another in this model.

At every time tick one of the animal's eight genes, selected at random, becomes "activated". A gene of a larger value has a greater probability of being selected. The activation

[51] As of this writing (September 2013) Palmiter's simulated evolution program is available for purchase at <http://lifesciassoc.home.pipeline.com/instruct/evolution/>.

of gene number k, $k = 0, \dots, 7$, makes the animal rotate in its place by an angle of $45 \times k$ degrees relative to its current orientation. After the rotation, the animal takes one step forward into the adjacent cell. Thus, an animal with a dominant gene 0 will tend to move forward along a straight line, while an animal with a dominant gene 2 will tend to go around in tight circles. An animal's genetic structure, therefore, affects the mode of its movements, which in turn affects its chances of finding food and survival.

An animal inherits its parent's genetic structure modulo small mutations. If the mutations are favorable to survival, then that animal and its descendants thrive. If the mutations are unfavorable, the descendants dwindle and die out. In the long run, this leads to a population which is increasingly more adept at survival.

Speciation, that is, the emergence of distinct species, occurs in response to environmental pressures. To demonstrate this, we set aside a small area of the world as a fertile region where food grows more plentifully than elsewhere, which is the desert, and where food is sparse.

Simulations show that in the long run two distinct classes of species emerge. One species consists of "desert dwellers" who are compelled by their dominant genes to travel over long distances, whereby increasing their chances of finding the sparsely scattered food in the desert. The others live in the fertile region and have dominant genes that limit their movements to small jitters, thereby keeping them within the fertile region where food is plentiful. As Dewdney puts it, "What normally is a disastrous genetic defect is actually an advantage in an overpopulated Garden of Eden."

Following Dewdney and Palmiter, I will refer to the fertile region as the "Garden of Eden" or simply the "Eden".

17.2 ▪ A more detailed description

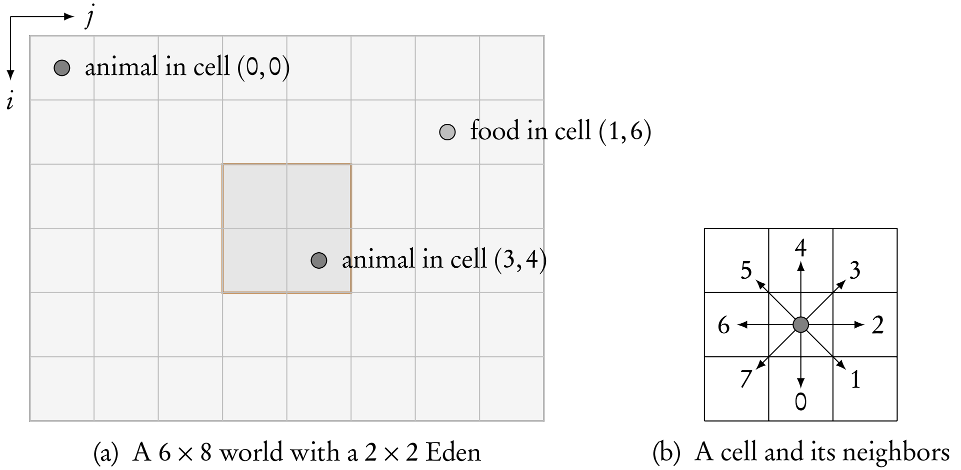
The previous section's brief sketch of the simulation's setup should give you a reasonably complete picture of the scenario which we intend to pursue. In the current section I will dwell on the setup's details. Thus, without further ado, let us proceed to the description of the simulation's components.

The world: The "world" is a rectangle whose opposite edges are identified; that is, if an animal leaves the bottom edge, it emerges from the corresponding spot on the top edge. The left and right edges are identified in a similar way.[52] This obviates the need for boundary conditions and thereby simplifies the model.

The world rectangle is discretized into an $h \times w$ grid of cells. The time is discretized into discrete time-steps. We refer to the cell in the world's ith row and jth column as the cell (i, j) in analogy to the customary addressing of the rows and columns of a matrix, but in contrast with matrices, our indices begin at zero. Figure 17.1(a) shows a (much too small) 6×8 world. A 100×100 world is much more interesting for simulations. Each cell has eight *neighboring cells*, numbered $0, \dots, 7$, as shown in the diagram of Figure 17.1(b). For a cell near an edge of the rectangle, neighboring cells wrap around to the opposite edge according to the world's toroidal structure.

In the future we will have occasions to refer to the world's "center" cell, which is a cell that is closest to the world rectangle's center. If h and w are odd, then the (i, j) index of the center cell is exactly $\big((h-1)/2, (w-1)/2\big)$. (You will have to draw a little picture to see that. Remember that cell indices begin with zero.) For general

[52] A topologist would say that this world is a *torus* or has a *toroidal topology*.



(a) A 6×8 world with a 2×2 Eden (b) A cell and its neighbors

Figure 17.1: The diagram on the left shows a (much too small) 6×8 world. There is a 2×2 Eden at the center. The i and j arrows at the top left are reminders that cells are indexed as in a matrix—from left to right in columns, from top to bottom in rows. The diagram on the right shows a generic cell and its eight neighboring cells, which are numbered 0 through 7.

h and w, we *define* the center cell by rounding the fractions to integers according to

$$\text{world's center cell} = \left(\left\lfloor \frac{h-1}{2} \right\rfloor, \left\lfloor \frac{w-1}{2} \right\rfloor \right). \tag{17.1}$$

The notation $\lfloor x \rfloor$ indicates the greatest integer less than or equal to x. For nonnegative integers p and q, the quotient p/q in C is evaluated to $\lfloor p/q \rfloor$ by default; therefore no special effort is required to evaluate the expression in (17.1).

The Eden: The *Eden* is a rectangular subset of the world which has a richer than normal food supply. The spatial inhomogeneity due to the presence of the Eden leads to *speciation*, that is, the emergence of genetically distinct species. Figure 17.1(a) shows a 2×2 Eden in a 6×8 world. *Setting one or both of Eden's dimensions to zero eliminates the Eden altogether.*

The world's toroidal structure implies that the Eden's location within the world is immaterial; all placements of the Eden are equivalent for the purposes of the simulation. The Eden's location, however, *does* affect the ease of some of the computations. Specifically, to interpret the simulation's results, it is essential, as we shall see later, to sort the animals according to their distances away from the Eden. Calculating such distances is easy if the Eden is located at or near the world's center cell since the concept of the distance then coincides with the usual notion of distance. If, however, the Eden is placed off center, then the distance calculations become slightly more involved due to the wrapping of the rectangle's edges. For this reason, I will assume from now on that the Eden is "centered" on the world rectangle as best as possible. Specifically, let the Eden be an $h' \times w'$ rectangle of cells. We position the Eden so that the cell at its northwest corner coincides with the world's

cell (i,j), where

$$(i,j) = \left(\left\lfloor \frac{h-1}{2} \right\rfloor, \left\lfloor \frac{w-1}{2} \right\rfloor\right) - \left(\left\lfloor \frac{h'-1}{2} \right\rfloor, \left\lfloor \frac{w'-1}{2} \right\rfloor\right). \tag{17.2}$$

Convince yourself that this centers the Eden in a best possible way relative to the world. Exact centering is not always possible since that depends on the parity of the world's and Eden's dimensions. Deviations from the exact center are insignificant if the world rectangle is large, as it is in our simulations.

When the Eden is positioned according to (17.2), then its "center" cell agrees—modulo the usual parity issues—with the world's center cell given in (17.1). We record this here for future reference:

$$\text{the Eden's center cell} = \left(\left\lfloor \frac{h-1}{2} \right\rfloor, \left\lfloor \frac{w-1}{2} \right\rfloor\right). \tag{17.3}$$

Food: At every time-step a morsel of food is dropped (or sprouts) in a randomly selected cell.[53] If an Eden exists, an *additional* morsel of food is dropped there. A food morsel contains a certain amount[54] of nutritional energy. A cell may hold more than one morsel of food.

Animals: Animals move about from cell to cell. Any number of animals may occupy a cell simultaneously. At every time-step every animal moves to a neighboring cell according to a mechanism that we shall describe. A move costs the animal 1 unit of energy. If the animal's energy drops to zero, it dies and is removed from the simulation. If it arrives at a cell that contains one or more morsels of food, it eats one morsel and gains energy.

Genes: An animal has a "chromosome" $\langle g_0, g_1, \ldots, g_7 \rangle$ consisting of eight integer-valued "genes", with $g_k \geq 1$ for all k. The chromosome is an intrinsic characteristic of an animal; it does not change during the animal's lifetime.

Movement: An animal may face in any of the eight possible directions numbered $0, 1, \ldots, 7$, as shown in Figure 17.1(b). At every time-step one of the animal's eight genes is randomly selected and "activated". The activation of the gene g_k prompts the animal to turn in its place by $45 \times k$ degrees counterclockwise, *relative to its current orientation.* It follows that if the animal is facing the direction d (where $d \in \{0, 1 \ldots, 7\}$), then after the activation of gene g_k it will be facing the direction $d + k \mod 8$.

After turning, the animal steps forward to the neighboring cell that it faces. This completes a movement cycle.

The probability of the selection of a gene is proportional to the gene's value. That is, a gene of larger value is more likely to be selected. For example, an animal with the chromosome structure $\langle 1, 3, 42, 2, 1, 3, 4, 2 \rangle$ will turn 90 degrees in almost every move since g_2 is the dominant gene.

Reproduction: If an animal's energy exceeds a certain level,[55] it splits in two in a manner of an asexual reproduction. The two halves are almost identical clones of the

[53] We assume uniform probability distribution; that is, one cell is as likely to receive a morsel as another.

[54] This is called `PlantEnergy` in our program.

[55] This is called `ReproductionThreshold` in our program.

original, except for (i) each one inherits half of the original's energy; and (ii) one of the two may inherit a slightly mutated chromosome, as described in the next paragraph.

Mutation: When copying a chromosome, we allow for a possible mutation by adding a randomly selected number from the set $\{-1, 0, 1\}$ to a randomly selected gene. (All random numbers here are uniformly distributed.) If the gene value drops below 1, we change it back to 1 because we don't want genes of 0 or negative values.[56]

This completes the description of our imaginary world. We initialize it with zero food and just one animal[57] with an arbitrary genetic structure but endowed with a largish initial energy; otherwise it's likely that it will die before it finds the first food morsel or has a chance to reproduce.

What becomes of the descendants of that animal in the long run? The principal tenet of the theory of evolution by natural selection is that mutations that are deleterious to survival wipe themselves out since, on the whole, animals with such mutations are outcompeted by the others; therefore they don't live long enough to reproduce. Conversely, mutations that are favorable to survival result in animals that successfully reproduce and bring forth descendants who are equally good, if not better, at survival.

Our simulations indicate that in a world without an Eden, the genetics of the population evolve so that the genes g_0, g_1, and g_7 gain dominance. These are the genes responsible for propelling the animal more or less in the forward direction. An animal that covers large territories has a better chance of encountering food than a sedentary animal that hovers in a small area waiting for food morsels to drop in its lap.

With the scenario described above, the animal population remains more or less homogeneous because probabilistically each animal is like the other. Simulations show that the population evolves *en masse* toward a more fit species.

The situation is drastically different in the presence of the Eden. Simulations show that in the long run the animals diverge into two very distinct species:

The desert-dwellers have dominant g_0, g_1, and g_7 genes and consequently mostly move forward and cover large territories.

The Eden dwellers have dominant g_3, g_4, and g_5 genes and consequently hover in the Eden area to partake of its rich food supply.

It is a testimony to the robustness of the natural selection mechanism that a primitive model such as the one under consideration here exhibits the characteristic traits of evolution and speciation observed in the much more complex real world.

17.3 ▪ The *World Definition File*

Our program reads the simulation's parameters from a user-supplied *World Definition File* (*WDF*), a sample of which is shown in Listing 17.1. The *WDF* is read with the help of the `fetch_line()` function developed in Chapter 9 which strips away blank lines, comments, and leading and trailing whitespace from its input. The general form of a stripped-down *WDF* is

[56]Genes of value zero are not objectionable; they just will have zero probability of getting activated. It may be worthwhile to experiment with this alternative. I haven't.

[57]The program allows initialization with any number of animals, but starting with just one is more striking.

Listing 17.1: The *WDF world-no-eden.wdf* defines a world with only one animal and no Eden. The animal is provided with a good amount of initial energy to give it a chance to find food, and perhaps reach the reproduction threshold, before dying of starvation.

```
1  # A World Definition File for the evolution simulator
2  World 100 100       # height x width
3  Eden   0 0          # No Eden!
4  Plant Energy 80
5  Reproduction Threshold 300
6
7  # Animals:
8  ( 0  0)  0  200  |  5     5     5     5     5     5     5     5  |
```

```
1  World h w
2  Eden h w
3  Plant Energy e
4  Reproduction Threshold t
5  (n n) n n | n n n n n n n n |
6  ...
```

Line 1 gives the world's height and width, that is, the number of vertical and horizontal cells that make up the world rectangle, and line 2 gives the Eden's height and width. Lines 3 and 4 give the values of the plant and reproduction threshold energies. Line 5, which specifies the initial state of an animal, may be repeated any number of times (hence the "`...`" on line 6) to initialize the world with the desired number and kinds of animals. The `n`'s in that line stand for integers, not necessarily all the same. Its meaning is better explained through the explicit notation

$$(\ i \quad j\ )\ \ d \quad e\ \ |\ \ g_0 \quad g_1 \quad g_2 \quad g_3 \quad g_4 \quad g_5 \quad g_6 \quad g_7\ \ |,$$

where $(i\ j)$ is the animal's location (cell index) subject to $0 \le i < h$, $0 \le j < w$, where h and w are the world's height and width; $d \in \{0, 1, \dots, 7\}$ is the direction that it faces; e is its energy; and g_0 through g_7 are its gene values, where every $g_k \ge 1$.

Note that the *WDF* does not prescribe the Eden's location relative to the world rectangle. We are going to use the formula (17.2) for the Eden's placement.

17.4 ▪ The program's user interface

Our program reads the simulation's parameters from the `stdin`, runs the simulation, and writes the results to the `stdout`. If specifically requested, it also produces a number of graphical snapshots of the world which may be viewed individually, or in sequence as an animation as detailed in Sections 17.12 and 17.13.

Let us say that the program's executable file is named *evolution*. We invoke it through a command-line of the form

```
$ ./evolution options <world-no-eden.wdf >outfile.wdf
```

where *world-no-eden.wdf* is the *WDF* shown in Listing 17.1. The output, which is directed to a file named *outfile.wdf*, contains the state of the world after a prescribed number of time-steps. The format of the output conforms to the specifications of *WDF*; therefore

it can be used to reinitialize the simulation if you wish.[58] Of course you may name the input and output files as you wish; the program itself is unaware of those file names. It's your Unix shell that directs the data flow through the redirection symbols "<" and ">"; see Chapter 3 for an overview of the Unix shell.

The number of time-steps, or *updates* as we call it in the program, is specified in the `options` part of the command-line. If `options` is set to `1000000`, as in

```
$ ./evolution 1000000 <world-no-eden.wdf >outfile.wdf
```

the simulation will perform 1,000,000 updates. The largest possible number you may enter here is `ULONG_MAX`, which is defined in the standard header file *limits.h* and which is the largest **`unsigned long int`** that your C compiler can handle. On a typical computer nowadays it is likely to be one of $2^{32}-1 = 4,294,967,295 \approx 4.3 \times 10^9$ or $2^{64}-1 \approx 1.8 \times 10^{19}$. Beware that if you enter a number larger than `ULONG_MAX` on the command-line, it will be truncated to `ULONG_MAX` (silently!). In practice, however, there is not much reason for that many iterations; 10 million updates are more than enough for demonstrating evolutionary trends and speciation.

If you invoke the program as

```
$ ./evolution 1000000 20 <world-no-eden.wdf >outfile.wdf
```

it will perform the 1,000,000 updates as before; then it will take 20 snapshots of the world interleaved with 19 updates. The snapshots will be written as *Encapsulated PostScript* (EPS) files named *fig0000.eps*, ..., *fig0019.eps*. Only then it will write the *outfile.wdf* and exit.

Why only 19 updates? The sequence of operations consists of `S U S U ... S U S`, where `S` stands for a snapshot and `U` stands for an update. As you see, there is one less update than there are snapshots. It follows that the command

```
$ ./evolution 0 1 <world-no-eden.wdf >outfile.wdf
```

will capture a snapshot of the world's initial state which can be useful in itself.

If you enter unrecognizable options or no options at all, the program prints a brief usage note and exits:

```
$ ./evolution
Usage: ./evolution n [f]  <infile  >outfile
    n ≥ 0 : (required) number of updates
    f ≥ 0 : (optional) number of snapshots after n updates
    Reads World Definition from infile, performs
    n updates, and writes result to outfile
```

This message is produced through the `show_usage()` function, which is the subject of subsection 17.11.2.

17.5 ▪ The program's components

The program relies on Chapter 7's *xmalloc* module for allocating memory, Chapter 8's *array.h* for building vectors and matrices, Chapter 16's *linked lists* module for managing linked lists, and Chapter 9's *fetch-line* module for reading lines from a *WDF*. Therefore,

[58] Actually the output *does not* record the complete state of the world since our *WDF* format carries no information about which cells contain food. Consequently, the information about the food distribution is lost if you stop and restart the program. Part 17.4 of this chapter's *Projects* section suggests an extension to the program whereby the locations of the food morsels are recorded in a *WDF*.

Listing 17.2: The file *evolution.h*.

```
1  #ifndef H_EVOLUTION_H
2  #define H_EVOLUTION_H
3  #include "linked-list.h"
4  struct animal {
5      int i;                          // row number of animal's position
6      int j;                          // column number of animal's position
7      int d;                          // direction the animal is facing: 0,...,7
8      int e;                          // animal's energy
9      int genes[8];                   // the animal's chromosome (array of genes)
10 } animal;
11 struct world {
12     int world_h;                    // world's height
13     int world_w;                    // world's width
14     int eden_h;                     // Eden's height
15     int eden_w;                     // Eden's width
16     int plant_energy;               // plant energy
17     int reproduction_threshold;     // reproduction threshold
18     int **plants;                   // world_h × world_w array of plants
19     conscell *herd;                 // the head of the linked list of the animals
20 };
21 #endif /* H_EVOLUTION_H */
```

following the recommendations of Chapters 2 and 6, the program's directory will look like this:

```
$ cd evolution
$ ls -F
Makefile        fetch-line.h@   read.h                  write.c
array.h@        interlude.c     world-and-eden.wdf      write.h
evolution.c     linked-list.c@  world-no-eden.wdf       xmalloc.c@
evolution.h     linked-list.h@  world-to-eps.c          xmalloc.h@
fetch-line.c@   read.c          world-to-eps.h
```

The file named *world-and-eden.wdf* is just like *world-no-eden.wdf* of Listing 17.1 except that it has a 10×10 Eden. To get started with this project, make an *evolution* directory, create the files *world-no-eden.wdf* and *world-and-eden.wdf*, and establish the symlinks as shown above. In the rest of this chapter I will describe the contents of the remaining files.

17.6 ▪ The file *evolution.h*

The file *evolution.h*, shown in Listing 17.2, declares two structures, one to hold animal properties and the other to hold the world's properties. The associated comments should adequately explain the purpose of every member except, perhaps, the last two of `struct world`.

The `plants` member points to a matrix of size `world_h` × `world_w`. The (i, j) entry of that matrix is the number of food morsels in the world's cell (i, j). The `herd` member is the head of the linked list of the animals.

Since `struct world` refers to `conscell`, and since `conscell` is declared in the header file *linked-list.h*, we `#include` that header file on line 3.

Listing 17.3: An outline of the file *read.c*.

```
1  #include <stdio.h>
2  #include "xmalloc.h"
3  #include "fetch-line.h"
4  #include "linked-list.h"
5  #include "read.h"
6  #define BUFLEN 1024
7   ▶static int get_world_dimens(struct world *world,
8                              char *str, int lineno) ...
9   ▶static int get_eden_dimens(struct world *world,
10                             char *str, int lineno) ...
11  ▶static int get_plant_energy(struct world *world,
12                             char *str, int lineno) ...
13  ▶static int get_reproduction_threshold(struct world *world,
14                             char *str, int lineno) ...
15  ▶static struct animal *get_animal_specs(char *str, int lineno) ...
16  ▶static char *fetch_line_aux(char *buf, int buflen,
17                             FILE *stream, int *lineno) ...
18  ▶int read_wdf(struct world *world) ...
```

17.7 ▪ The files *read.[ch]*

Listing 17.3 shows an outline of the file *read.c* that defines a function `read_wdf()` (line 18) whose purpose is to read a *WDF* and parse its contents. All the other functions in *read.c* are for `read_wdf()`'s private use; therefore they are declared **`static`** to make them invisible to the outside world. (See Section 1.6 for an explanation of the **`static`** specifier.) In the following subsections I will describe the roles of the individual functions that appear in Listing 17.3.

The function `read_wdf()` repeatedly calls the function `fetch_line()` (from the *fetch-line* module of Chapter 9) to extract nonempty trimmed lines from a *WDF* that arrives from the `stdin`. According to the conventions of a *WDF* (see Section 17.3), the first four such lines are expected to be the specifications of the world's and the Eden's dimensions and the plant and reproduction threshold energies. These may be followed by any number of animal specification lines. Altogether, there are five distinct types of lines that we expect to encounter; therefore we write five distinct function to handle those types. These are the functions

```
get_world_dimens()        get_reproduction_threshold_energy()
get_eden_dimens()         get_plant_energy()
get_animal_specs()
```

that appear on lines 7 through 15 of Listing 17.3. I will describe the details of these and the related functions in the following subsections.

17.7.1 ▪ The function `get_world_dimens()` and friends

The function `get_world_dimens()` that appears on line 7 of Listing 17.3 is responsible for parsing the first line of a *WDF*, which is expected to be a string of the form `"World h w"`, where the integers `h` and `w` are the world's height and width. If the string is properly formed, `get_world_dimens()` extracts the numbers `h` and `w`, stores them in the `world_h` and `world_w` members of the **`struct`** `world`, and returns 1 to

Listing 17.4: The function `get_world_dimens()` in the file *read.c*.

```
1  static int get_world_dimens(struct world *world, char *str, int lineno)
2  {
3      if (sscanf(str, "World %d %d",
4                  &world->world_h, &world->world_w) == 2)
5          return 1;
6      else {
7          fprintf(stderr, "stdin:line %d: expected to find "
8                  "World dimensions here\n", lineno);
9          return 0;
10     }
11 }
```

indicate success. Otherwise the line is garbled; therefore it prints a diagnostic on the `stderr` and returns 0 to indicate failure. Listing 17.4 shows my implementation of `get_world_dimens()`. The function receives a pointer to **struct** `world` where it will store the world's height and width; a pointer to the string to parse; and the line number of the string as it was read from the `stdin`. The error diagnostic prints that line number to help the user locate the trouble spot.

The standard library function `sscanf()` returns the number of successfully matched and assigned items in the call. Therefore, if the input string is properly formed, we expect `sscanf()` to return 2. That's what we are checking on line 3 of Listing 17.4.

I will let you write the functions `get_eden_dimens()`, `get_plant_energy()`, and `get_reproduction_threshold_energy()` that appear in Listing 17.3. These are close variants of `get_world_dimens()`. Add your implementations to *read.c*.

17.7.2 ▪ The function `get_animal_specs()`

The purpose of the function `get_animal_specs()` that appears on line 15 of Listing 17.3 is to read the specifications of one animal in a *WDF*. (Refer to the sample *WDF* in Listing 17.1 on page 166 regarding animal specifications.) As such, it is very similar to the previous subsection's `get_world_dimens()` function. Nevertheless, I have shown the entire function in Listing 17.5 because there is a slight twist.

The call to `sscanf()` works exactly as before, but it is reading 12 integers now. If `sscanf()` returns other than 12, `get_animal_specs()` prints the usual diagnostic message and returns `NULL` to indicate failure. Otherwise it stores the 12 numbers in a **struct** `animal`. This is where the new aspect comes in; the structure has to be created. So it calls `xmalloc()` to allocate memory for a **struct** `animal` and populates it with the values retrieved by `sscanf()`. Finally it returns a pointer to the structure to the calling function. Note that unlike the previous section's `get_*()` functions that return **int**, this one returns a pointer to a **struct** `animal` on success or `NULL` on failure.

17.7.3 ▪ The function `fetch_line_aux()`

The function `fetch_line_aux()` that appears on line 16 of Listing 17.3 is a helper function which plays no deep role but is quite convenient for our purposes. Recall that the function `fetch_line()` which we developed in Chapter 9 returns `NULL` when the input is exhausted. In the context of reading a *WDF*, if the end of the input occurs *after* all the required data is in, the situation is normal; we *expect* that to happen. If, however, the end of the input occurs *before* all the world's parameters are retrieved,

Listing 17.5: The function `get_animal_specs()` in file *read.c*

```
1  static struct animal *get_animal_specs(char *str, int lineno)
2  {
3      struct animal *animal;
4      int i, j, d, e, genes[8], r;
5      r = sscanf(str, "( %d %d ) %d %d | %d %d %d %d %d %d %d %d |",
6              &i, &j, &d, &e,
7              &genes[0], &genes[1], &genes[2], &genes[3],
8              &genes[4], &genes[5], &genes[6], &genes[7]);
9      if (r == 12) {
10         animal = xmalloc(sizeof *animal);
11         animal->i = i;
12         animal->j = j;
13         animal->d = d;
14         animal->e = e;
15         for (int k = 0; k < 8; k++)
16             animal->genes[k] = genes[k];
17         return animal;
18     } else {
19         fprintf(stderr, "stdin:line %d: expected to find an "
20                 "animal description here\n", lineno);
21         return NULL;
22     }
23 }
```

Listing 17.6: The function `fetch_line_aux()` in file *read.c* is a simple wrapper around `fetch_line()`.

```
1  static char *fetch_line_aux(char *buf, int buflen,
2                              FILE *stream, int *lineno)
3  {
4      char *s = fetch_line(buf, buflen, stream, lineno);
5      if (s == NULL)
6          fprintf(stderr, "stdin:line %d: premature end of input\n",
7                  *lineno);
8      return s;
9  }
```

there is cause for alarm; usually that's an indication that the *WDF* is malformed. The `NULL` returned by `fetch_line()` does not distinguish between these two cases. The function `fetch_line_aux()`, whose implementation is shown in Listing 17.6, acts just like `fetch_line()` but prints a diagnostic message to `stderr` when the end of input is reached. We call the plain `fetch_line()` when "no more input" is an acceptable outcome and call `fetch_line_aux()` when it is not. Add `fetch_line_aux()` to your *read.c*.

17.7.4 ▪ The function `read_wdf()`

We have all the necessary ingredients now to put together the function `read_wdf()` that appears on line 18 of Listing 17.3 on page 169 and whose purpose was outlined in the beginning of the current section.

Listing 17.7: A tentative sketch of the function `read_wdf()`.

```
1  int read_wdf(struct world *world)          // tentative
2  {
3      char buf[BUFLEN], *s;
4      int lineno = 0;
5      s = fetch_line_aux(buf, BUFLEN, stdin, &lineno);
6      get_world_dimens(world, s, lineno);
7  }
```

Here is the plan: We call `fetch_line_aux()` to skip over comments and blanks of the *WDF*, and grab the first significant line, which is expected to define the world's dimensions. We pass that line to `get_world_dimens()` to retrieve the world's height and width. Listing 17.7 gives a tentative sketch. We initialize `lineno` to zero there because at the program's startup we haven't read any lines. On line 5 we pass `buf`[59] to `fetch_line_aux()`, which it will use to store the lines that it reads from the `stdin`. Note that we call `fetch_line_aux()` rather than `fetch_line()` because hitting the end of the input at this stage is premature and a diagnostic message should be printed if it occurs. We pass the *address* of `lineno` to `fetch_line_aux()` so that it may access that memory location to increment the value stored there as it reads lines from the `stdin`. If successful, `fetch_line_aux()` returns a pointer to a trimmed string—we call it `s`—where we should look for useful data. In line 6 we pass the pointer `s` to `get_world_dimens()` for parsing. If successful, `world`'s `world_h` and `world_w` members will be populated by the read data.

The statements in the previous two paragraph were qualified by "if successful". What would happen otherwise?

If `fetch_line_aux()` fails, it returns `NULL`. If `get_world_dimens()` fails, it returns `0`. In C's Boolean algebra both of these are interpreted as *false*. Pointers other than `NULL` and numbers other than `0` are interpreted as *true*. Consequently, the compound expression

```
(s = fetch_line_aux(buf, BUFLEN, stdin, &lineno)) &&
                                   get_world_dimens(world, s, lineno)
```

is *true* if and only if both function calls are successful. We apply the same logic to evaluating the outcomes of calls to `get_eden_dimens()`, `get_plant_energy()`, and `get_reproduction_threshold()`. Altogether, these necessitate the checking of the successes of eight function calls:

```
(s = fetch_line_aux(buf, BUFLEN, stdin, &lineno))  &&
get_world_dimens(world, s, lineno)                 &&
(s = fetch_line_aux(buf, BUFLEN, stdin, &lineno))  &&
get_eden_dimens(world, s, lineno)                  &&
(s = fetch_line_aux(buf, BUFLEN, stdin, &lineno))  &&
get_plant_energy(world, s, lineno)                 &&
(s = fetch_line_aux(buf, BUFLEN, stdin, &lineno))  &&
get_reproduction_threshold(world, s, lineno)
```

This compound expression evaluates to *true* if and only if each of the eight function calls are successful.

[59]This decays to a pointer to the first element of the array `buf[]`. See Section 4.4 for the meaning of this.

Listing 17.8: The function `read_wdf()` in the file *read.c*. It reads a *WDF* and return 1 on success and `0` on failure.

```
int read_wdf(struct world *world)
{
    int animal_count = 0;
    char buf[BUFLEN];
    struct animal *animal;
    int lineno = 0;
    char *s;

    int result =
        (s = fetch_line_aux(buf, BUFLEN, stdin, &lineno))  &&
        get_world_dimens(world, s, lineno)                 &&
        (s = fetch_line_aux(buf, BUFLEN, stdin, &lineno))  &&
        get_eden_dimens(world, s, lineno)                  &&
        (s = fetch_line_aux(buf, BUFLEN, stdin, &lineno))  &&
        get_plant_energy(world, s, lineno)                 &&
        (s = fetch_line_aux(buf, BUFLEN, stdin, &lineno))  &&
        get_reproduction_threshold(world, s, lineno);

    if (!result)
        return 0;

    while ((s = fetch_line(buf, BUFLEN,
                    stdin, &lineno)) ≠ NULL) {
        if ((animal = get_animal_specs(s, lineno)) == NULL)
            return 0;
        else {
            world→herd = ll_push(world→herd, animal);
            animal_count++;
        }
    }

    fprintf(stderr, "# %d animal%s read from the input\n",
            animal_count, animal_count == 1 ? "" : "s");

    return 1;
}
```

The entire function `read_wdf()` is shown in Listing 17.8. Lines 9 through 17 assign the value of the compound statement of the eight function calls to the variable `result`. On line 19 we check the value of `result`; if it is *not true*, then at least one of the eight function calls has failed. In that case we return from `read_wdf()` with a status value of `0` to indicate failure. The calling function will have to deal with the consequences. There is no need to print anything here because each of the eight functions prints its own diagnostics when an error occurs.

Remark 17.1. Tacit in the logic detailed above is what is known as the *short-circuiting property* of C's Boolean operators. Specifically, C evaluates compound statements such as `A && B && C && ⋯` from *left to right*. It stops the evaluation chain as soon as the outcome of the overall statement becomes evident. For instance, if `A` is *false*, then the

Listing 17.9: The header file *read.h*.

```
1 #ifndef H_READ_H
2 #define H_READ_H
3 #include "evolution.h"
4 int read_wdf(struct world *world);
5 #endif /* H_READ_H */
```

entire compound statement is *false* regardless of the values of B and C; therefore B and C are not evaluated at all. We often rely on the short-circuiting property in very crucial ways, and what we have in the `result = ···` construction is no exception. Note that if the call to the first `fetch_line_aux()` fails, `s` will be `NULL`. If the next expression, that is, the call to `get_world_dimens()`, were to be evaluated, the program would crash because there is no provision in `get_world_dimens()` for handling a `NULL` input. But we need not be concerned about that since the `NULL` outcome of the first call short-circuits the computation and `get_world_dimens()` is never called.

Lines 22–30 of `read_wdf()` handle the animal specification lines. The input may specify any number of animals; therefore we process those through a **while**-loop. We stop when `fetch_line()` returns `NULL`, which signals that there is no more data to read. Note that we call `fetch_line()` rather than `fetch_line_aux()` because reaching the end of the input at this stage is normal.

The string fetched on line 22 is passed to `get_animal_specs()` on line 24. If `get_animal_specs()` returns `NULL`, it indicates that the string is malformed. Then we return from `read_wdf()` with a status of 0 to indicate failure. Otherwise, we have received a pointer to a valid **struct** `animal`; therefore on line 27 we call `ll_push()` to insert that animal in the linked list of animals and then increment `animal_count`. The animal count does not have a great significance; it's sent to the `stderr` on line 32 as an information item to reassure the user that the expected number of lines were read. Finally, `read_wdf()` returns with a status of 1 to indicate the successful completion of its task.

This completes the description of the file *read.c*. We add a companion *read.h* header file, shown in Listing 17.9, that presents *read.c*'s interface. Since the function `read_wdf()` is the only one in *read.c* with external linkage—all others are **static**—then `read_wdf()` is the sole function declared in *read.h*. On line 3 we **#include** *evolution.h* since it contains the declaration of "**struct** `world`", which is referenced on line 4.

17.8 ▪ The files *write.[ch]*

The file *write.c* defines a function `write_wdf()` whose purpose is to write a report of the current state of the world. The report's format conforms to the specifications of a *WDF*, making it possible to use it as input to the evolution simulator for a continuation of the simulation. Listing 17.10 gives an outline of *write.c*, and Listing 17.11 shows a sample report. The report's top half is produced by simply writing out the information that is available in the **struct** `world`. The listing of the animals in the bottom half is produced through a call to the function `print_herd()` (line 4 of Listing 17.10), which receives a pointer to the linked list of animals, walks through the list, and prints the information in each node. To simplify its work, `print_herd()` delegates the printing of each animal line to a helper function, `print_animal()` (line 3 of Listing 17.10), which

Listing 17.10: An outline of the file *write.c*.

```
1 #include <stdio.h>
2 #include "write.h"
3 ▶static void print_animal(struct animal *animal)
4 ▶static void print_herd(conscell *herd)
5 ▶void write_wdf(struct world *world)
```

Listing 17.11: A sample report produced by the function `write_wdf()` in the file *write.c*. The "`Math 625`" is the course number for which this book was written. Change it to something that's more relevant to you.

```
# Machine-generated by the Math 625 Evolution Simulator

World 60 100  # h x w
Eden 10 15   # h x w
Plant Energy 80
Reproduction Threshold 300

#( i   j)  d     e  | g[0] g[1] g[2] g[3] g[4] g[5] g[6] g[7]|
#-------------------|---------------------------------------|
( 84  39)  6    98  |  10    2    4    8    3    6    5    7 |
( 36  51)  7   117  |   7    8    2    7    1    3    2    9 |
( 49  97)  2    94  |   6    8    5    9    3    4    9   10 |
( 23  97)  7   106  |   3    6    4    8    6    7    6    1 |
( 43  93)  7    86  |   3    8    7    4    7    2    5    9 |
( 82  33)  1   113  |   4    7   10    6    7    9    5   10 |
( 39  81)  5   116  |   5    3   10   10    2    9    7    5 |
( 61  28)  6    19  |   5    3    2    3    6    5    2   10 |
# animal count: 8
```

receives a pointer to an animal structure and formats and prints its data. You should be able to complete the file *write.c* yourself, so I won't tell more about it. You will supplement this with a header file *write.h* that presents *write.c*'s interface.

17.9 ▪ The files *world-to-eps.[ch]*

I have written a function named `world_to_eps()` with the prototype

```
void world_to_eps(struct world *world, char *epsfile);
```

which receives a pointer to a **`struct`** `world`, from which it extracts the information necessary for writing a graphical image of the world, in the EPS format, into a file whose name is specified in the `epsfile` argument. The result depicts the world rectangle in a white background, the Eden in a pale green background, the food morsels as green dots, and the animals as red dots. In the monochrome rendering in Figure 17.2, the Eden, the food, and the animals appear as a gray rectangle, light gray dots, and dark gray dots, respectively.

Implementing the function `world_to_eps()` is not a part of this project because that requires some familiarity with the *PostScript* language, which I don't want to make a prerequisite for this book. You should download my implementation of the files

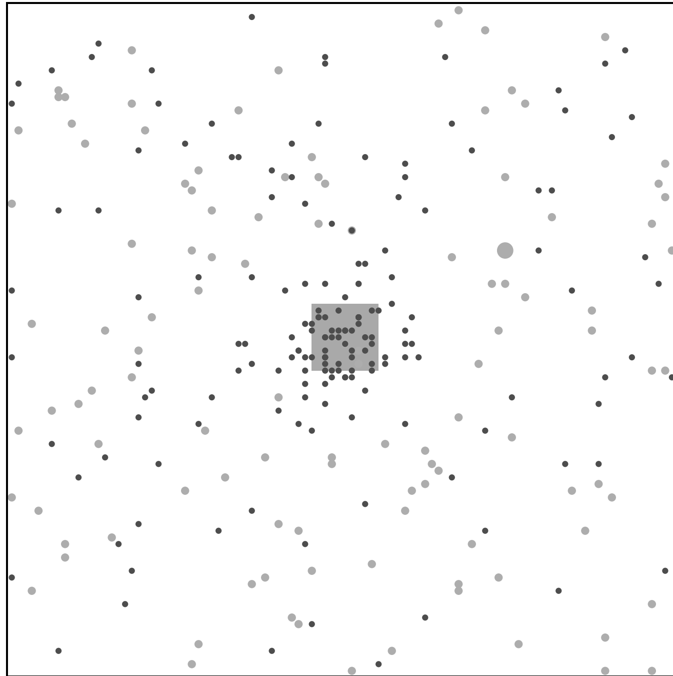


Figure 17.2: A sample *PostScript* image produced through the command "`./evolution 1000000 1 < world-and-eden.wdf`". The small square near the center is the Eden.

world-to-eps.c and *world-to-eps.h* from this book's website, drop them in this project's directory, and compile them along with the rest of your files.

17.10 ▪ Interlude (and a mini-project)

We are at about the halfway mark through *Project Evolution*. It's a good idea to stop for a moment and test what we have thus far.

Your assignment is to write a program, say *interlude.c*, that calls `read_wdf()` to read a *WDF* from the `stdin` and then calls `write_wdf()` to print a report to the `stdout`.

(a) Compile the program into an executable, say *interlude*; then run it as

```
$ ./interlude <world-no-eden.wdf
```

where *world-no-eden.wdf* is given in Listing 17.1 on page 166. If all is well, this should print out the contents of the input file, minus comments and extra whitespace, and exit.

(b) Copy *world-no-eden.wdf* to a temporary file, say *tmp.wdf*. Insert a few more animals in it, and try

```
$ ./interlude <tmp.wdf
```

(c) Introduce deliberate errors of various sorts in *tmp.wdf*, and verify that your program catches them.

Listing 17.12: An outline of the file *evolution.c*.

```
1  #include <stdio.h>
2  #include "evolution.h"
3  #include "xmalloc.h"
4  #include "array.h"
5  #include "read.h"
6  #include "write.h"
7  #include "linked-list.h"
8  #include "world-to-eps.h"
9  #define MAX(a,b)  ((a) > (b) ? (a) : (b))
10 struct point {
11     int i;
12     int j;
13 };
14 ▶static inline int random(int n) ...
15 ▶static void free_herd(conscell *herd) ...
16 ▶static int dead_or_alive(const void *aa) ...
17 ▶static conscell *remove_the_dead(conscell *herd) ...
18 ▶static int nearer_the_eden(
19             const void *aa, const void *bb, void *params) ...
20 ▶static void initialize_plants(struct world *world) ...
21 ▶static void add_plants(struct world *world) ...
22 ▶static int gene_to_activate(int genes[8]) ...
23 ▶static void turn(struct animal *animal) ...
24 ▶static void move(struct world *world, struct animal *animal) ...
25 ▶static void feed(struct world *world, struct animal *animal) ...
26 ▶static struct animal *clone(struct animal *old) ...
27 ▶static void mutate(int genes[8]) ...
28 ▶static void reproduce(
29                 struct world *world, struct animal *animal) ...
30 ▶static void update_world(struct world *world) ...
31 ▶static void evolve(struct world *world, unsigned int n) ...
32 ▶static void evolve_with_figs(
33                 struct world *world, unsigned int n) ...
34 ▶static void show_usage(char *progname) ...
35 ▶int main(int argc, char **argv) ...
```

Remark 17.2. Your file *interlude.c* will be just a few lines long; don't make a big production of it. If you need help with getting started, have a peek at Listing 17.13 on page 178, which treats a *much more complex* case. All you need are the simplified versions of lines 5, 6, 26, 38, and 43.

17.11 ▪ The file *evolution.c*

The core of the evolution simulator is in the file *evolution.c*. This somewhat longish file consists of many short and simple functions. Listing 17.12 provides an outline. The best place to begin the description of its contents is at the function `main()` that appears on line 35, so that's what we will do next.

Listing 17.13: The function `main()` in file *evolution.c*.

```
int main(int argc, char **argv)
{
    unsigned long int n;           // number of updates
    unsigned long int f = 0;       // number of figures to generate
    struct world World;
    struct world *world = &World;
    int exit_status = EXIT_FAILURE;
    world→herd = NULL;
    world→plants = NULL;

    if (argc < 2 || argc > 3) {
        show_usage(argv[0]);
        goto cleanup;
    }

    if (sscanf(argv[1], "%lu", &n) ≠ 1) {
        show_usage(argv[0]);
        goto cleanup;
    }

    if (argc > 2 && sscanf(argv[2], "%lu", &f) ≠ 1) {
        show_usage(argv[0]);
        goto cleanup;
    }

    if (!read_wdf(world))
        goto cleanup;

    initialize_plants(world);
    evolve(world, n);
    if (f > 0)
        evolve_with_figs(world, f);

    struct point eden_center;
    eden_center.i = (world→world_h - 1)/2;
    eden_center.j = (world→world_w - 1)/2;
    world→herd = ll_sort(world→herd, nearer_the_eden, &eden_center);
    write_wdf(world);
    exit_status = EXIT_SUCCESS;

cleanup:
    free_matrix(world→plants);
    free_herd(world→herd);
    return exit_status;
}
```

17.11.1 ▪ The function `main()`

Listing 17.13 shows the function `main()` in its entirety. It is the longest of the functions in *evolution.c*; the routine parsing of the command-line arguments is partly responsible for that length. To explain the several new features introduced in this function, I will go through it in detail. But before that, let me comment on its overall design.

The execution of the function `main()` may have to be halted at various points due to possible faulty data submitted by the user. For instance, the user may have given illegal command-line arguments, or perhaps the *WDF* is incomplete or garbled. Whenever any such error is encountered within `main()`, the program's flow jumps via a **`goto`** to the `cleanup` label on line 41, subsequent to which the memory resources are freed and the program terminates with the exit status assigned to a variable named `exit_status`,[60] which is assigned the default value of `EXIT_FAILURE` on line 7. Only when the program nears the completion of its mission on line 39 is its value changed to `EXIT_SUCCESS`.

Here we begin the line by line analysis of the function `main()`:

Line 1: We invoke `main()` with the `argc` and `argv` parameters because we intend to capture the command-line arguments. Here you will need the information from Section 4.7 on the details of how the command-line arguments are stored in memory.

Lines 3 and 4: The user specifies the number of iterations and the number of snapshots on the command-line. The former is required; the latter is optional. Further down, where we parse the command-line arguments, we store those numbers in the variables `n` and `f`, respectively. We set `f` to zero here as its default value.

Line 5: The `World` defined here is a **`struct`** `world` which is to hold the complete data that characterizes the world at every instant. Most of our program's functions need access to that data to perform their tasks. It suffices to pass `World`'s address to such functions. They will query the structure for data through that address, and even modify that data if they are supposed to.

Line 6: We define the auxiliary variable `world` (with lowercase `w`) to hold `World`'s address. This is by no means essential since we may equally well use `&World` for `World`'s address. There is a slight drawback to that, however. Within `main()` we have access to both `World` and its address, `&World`. Outside of `main()` we have access only to `World`'s address. Working with `World` in `main()` and `World`'s address outside `main()` results in a certain cognitive dissonance that we wish to avoid. That's why we work with `world` instead of `&World` in the rest of `main()`.

Line 7: Upon exit, the program will return the value of `exit_status` to the calling environment. We set `exit_status` to `EXIT_FAILURE` as its default value. Only if all is well will we change it to `EXIT_SUCCESS`.

Lines 8 and 9: The `world→herd` points to the head of the linked list of animals. We set it to `NULL` here to indicate that initially the linked list is empty.

The `world→plants` is going to point to a `world_h` × `world_w` matrix (to be allocated later) whose (i, j) entry holds the number of plants in cell (i, j). We set it to `NULL` here for a different, and a more subtle, reason.

In several places in `main()` where we encounter unacceptable situations, we bail out through a "**`goto`** `cleanup`", which makes the execution jump to the `cleanup` label where `free_matrix(world→plants)` is called, among other things, to free the memory associated with the plants matrix. If we omit line 9, then `world→plants` will have an undetermined (junk) value; therefore the call to `free_matrix(world→plants)` is likely to crash the program. On the other

[60] There is nothing special about the name "`exit_status`". I called it that because it expresses its intent well.

hand, if `world→plants` has been set to `NULL`, then calling `free_matrix()` is harmless since it is designed to handle a `NULL` argument gracefully. This may be a good time to go back and have a look at `free_matrix()`'s code in Chapter 8.

Line 11: The next dozen lines or so extract the user-specified options from the command-line arguments. To appreciate what happens here, you will have to compare the code in `main()` with the layout of the command-line arguments shown in Figure 4.1 on page 26.

As we saw in Section 17.4, our program is expected to be invoked as "`progname n`" or "`progname n f`"; therefore we expect `argc` to be either 2 or 3. If on line 11 we discover that `argc` is less than 2 or more than 3, we print a "usage message" to remind the user what needs to be done and then quit the program. The function `show_usage()` will be described later, but note that we pass `argv[0]` to `show_usage()` so that it can compose a coherent message complete with the program's name.

Line 16: If we arrive here, then `argc` is at least 2. We apply `sscanf()` to the `argv[1]` to extract its numerical value into `n`. If the extraction is successful, `sscanf()` will return 1; otherwise the input is not an **`unsigned long int`**, so we call `show_usage()` to alert the user, and then we exit the program.

Line 21: If `argc > 2`, then a third item is present on the command-line. We apply `sscanf()` to `argv[2]` to extract its numerical value into `f`. Again, if the extraction fails, we call `show_usage()` and then exit the program.

Remark 17.3. What happens if the user enters `-1` for `n` or `f` on the command-line? Does that set `n` or `f` to `-1`? No! Since `n` and `f` are of type **`unsigned long int`**, and due to the modular arithmetic of C's unsigned types, that `-1` is received as `ULONG_MAX`, which is the largest value that an **`unsigned long int`** can hold.

Remark 17.4. `ULONG_MAX` $= 2^{32} - 1 = 4,294,967,295$ on my computer. What happens if I enter something larger, say $4,294,967,800$, on the command-line? Is the number reduced according to modular arithmetic? No! The C standard is quite explicit about the behavior of `sscanf()` in that circumstance; if the specified number is too large, it is truncated to `ULONG_MAX`.[61]

Remark 17.5. What happens if the user enters `1000xyz` for `n`? According to the semantics of `sscanf()`, `n` is set to `1000` and the trailing junk is silently discarded. If you wish to alert the user about such trailing junk, you should use the standard library's `strtoul()` instead of `sscanf()`. See Chapter 5 for details.

Line 26: We call the function `read_wdf()` (defined in file *read.c*) to read the user-supplied *WDF* from the `stdin`. We pass `world`, which is the address of `World`, to `read_wdf()`. That's all it needs to do its work.

If `read_wdf()` encounters an error in the *WDF*—garbled text or incomplete data, for instance—it returns *false* (actually 0). If all is well, it returns *true* (actually 1).

[61] Furthermore, the standard library's `errno` variable is set to the macro `ERANGE`, which indicates a range error. You may print a text version of that out-of-range message through the standard library's `perror()` function if you want.

On line 26 we check the return value. If it's other than *true*, we jump to `cleanup` and therefore exit the program. No message is printed here because `read_wdf()` itself prints diagnostics as it encounters problems.

Line: 29: The function `initialize_plants()` allocates memory for the matrix of plants and sets all its elements to zero. The world begins with no plants at all.

Line: 30: This is where the real action takes place—`evolve(world, n)` takes the world through `n` time-steps.

Line 31: If `f` is nonzero, then the program takes `f` graphical snapshots of the world, interleaving world updates with snapshots.

Lines 34–37: We define `eden_center` according to the formula (17.3) on page 164. The **`struct`** `point` structure that occurs on line 34 is declared on line 10 in Listing 17.12. We call `ll_sort()` to sort the linked list of the animals according to increasing distance from `eden_center`. Thus, animals nearer the Eden will come first in the list, and a listing of the herd should reveal genetic differences, if any, between the animals near and far from the Eden. The comparison function, `nearer_the_eden()`, that appears as an argument to `ll_sort()` on line 37 is described in subsection 17.11.6 on page 182.

Remark 17.6. The placement of the declaration of `eden_center` on line 34 assumes C99. To revert to C89, move that declaration to the top of `main()` since C89 requires all declarations to precede executable statements in a block.

Line 38: We write the world's current data in the *WDF* format to the `stdout`.

At this point the program has accomplished its mission; therefore on the next line we change the value of `exit_status` to `EXIT_SUCCESS`.

Line 41: The `cleanup` label has been the jumping target from many places throughout `main()`. There is not much to do once we arrive here. We free the memory resources and return from `main()`—and consequently exit the program—with the status of `exit_status`, which has been set earlier to one of `EXIT_SUCCESS` or `EXIT_FAILURE`.

This completes the description of the somewhat longish function `main()`. In comparison, the remaining functions in *evolution.c* are shorter and simpler. We set out to describe those next.

17.11.2 ▪ The function `show_usage()`

The function `show_usage()` that appears on line 34 in Listing 17.12 on page 177 prints a brief help message on the program's usage. It is invoked from several places in `main()` (in Listing 17.13) wherever the user's input disagrees with the program's expectations. I will not give an outline; you write the whole thing. My version prints

```
Usage: ./evolution n [f]  <infile  >outfile
    n ≥ 0 : (required) number of updates
    f ≥ 0 : (optional) number of snapshots after n updates
    Reads World Definition from infile, performs
    n updates, and writes result to outfile
```

Remark 17.7. The program's name arrives as an argument to `show_usage()`. Use that; don't hard-code the program's name into the function's body.

Remark 17.8. On the "Usage" line I have placed `f` in square brackets. That's the traditional way—at least in the Unix world—of indicating that the argument is optional.

17.11.3 ▪ The function `random()`

The function `random()` that appears on line 14 in Listing 17.12 on page 177 is exactly the function `random()` of Listing 10.1 on page 73 in Chapter 10. Just copy the code from there into your *evolution.c*.

17.11.4 ▪ The function `free_herd()`

The function `free_herd()` that appears on line 15 in Listing 17.12 on page 177 frees the memory resources associated with a linked list of animals. Note that the function `ll_free()` developed in Chapter 16 merely frees the memory of a linked list's *cons cells*. It does not touch the data that the cons cells point to. Freeing the linked list of animals, therefore, requires two passes, first to free the data and then to free the cons cells, as in

```
1  static void free_herd(conscell *herd)
2  {
3      for (conscell *p = herd; p != NULL; p = p->next)
4          free(p->data);
5      ll_free(herd);
6  }
```

17.11.5 ▪ The functions `dead_or_alive()` and `remove_the_dead()`

After each time-step, the program scans the linked list of the animals and removes the dead. An animal is dead if its energy has dropped to zero or below. The function `dead_or_alive()` that appears on line 16 in Listing 17.12 on page 177 receives a pointer to an animal and returns *true* or *false* (that is, 1 or 0) depending on whether the animal is alive or dead, respectively. I will leave it to you to implement that function.

The function `remove_the_dead()` that appears on line 17 in Listing 17.12 is a simple wrapper around Chapter 16's `ll_filter()` function to cull the herd. Here it is in its entirety:

```
1  static conscell *remove_the_dead(conscell *herd)
2  {
3      conscell *dead = NULL;
4      herd = ll_filter(herd, dead_or_alive, &dead);
5      free_herd(dead);
6      return herd;
7  }
```

Be sure to understand how it works, and then add it to your *evolution.c*.

17.11.6 ▪ The function `nearer_the_eden()`

The most striking and significant finding of this simulated evolution is that in the presence of the Eden, the animals evolve into two distinct species: those who live in or near the

Eden and those who live elsewhere. The two species are distinguished by their distinct genetic structures. A good way of bringing out that distinction is to sort the linked list of the animals according to their distances away from the center of the Eden. This puts the species of one kind near the beginning of the list and the species of the other kind near the end of the list.

The sorting is performed through the function `ll_sort()`, as explained on page 181. That function, in turn, relies on a helper function, `nearer_the_eden()`, to determine the order of the items in the linked list. The latter appears on line 18 in Listing 17.12 (page 177). We see that it is invoked with three arguments. The first two are pointers to **`struct`** `animal`, and the third is a pointer to a **`struct`** `point`, which is declared on line 10 in Listing 17.12 on page 177 and which holds the indices (e_i, e_j) of the Eden's center. The function `nearer_the_eden()` computes the distances of the two animals from (e_i, e_j), and returns `-1`, `0`, or `1` depending on whether the first animal is closer to, at an equal distance from, or farther from the Eden, compared to the second. Distance may be measured in any meaningful way. The easiest (and possibly the most relevant) metric measures the distance between the cells (a_i, a_j) and (e_i, e_j) through the formula

$$\operatorname{dist}\big((a_i,a_j),(e_i,e_j)\big) = \max\big(|a_i - e_i|, |a_j - e_j|\big),$$

where the max function produces the larger of its two arguments. You may use the preprocessor macro `MAX()` defined on line 9 of Listing 17.12 for this purpose.

You have all the necessary bits now to implement the function `nearer_the_eden()` and add it to your *evolution.c*. It will be a slight variant of the function `cmp2()` of page 155.

Remark 17.9. Implicit in the distance formula above is the assumption that the Eden is located near the center of the world rectangle. (See the discussion on page 163.) If the Eden were located significantly off center, that formula wouldn't apply since it does not account for the wrapping around of the world's edges. For instance, if the Eden is near one of the edges of the world rectangle, then a point on the opposite edge can be actually quite close to the Eden, while the formula above says otherwise. Should you wish to modify the program to allow for an off-center Eden, replace that distance formula with

$$\operatorname{dist}\big((a_i,a_j),(e_i,e_j)\big) = \max\big(\phi(a_i - e_i, h), \phi(a_j - e_j, w)\big),$$

where h and w are the world's height and width, and the function ϕ is defined as

```
int phi(int x, int L)
{
    int d = abs(x);
    if (d > L/2)
        d = L - d;
    return d;
}
```

17.11.7 ▪ The function `initialize_plants()`

The function `initialize_plants()` that appears on line 20 in Listing 17.12 (page 177) calls `make_matrix()` to create the $h \times w$ matrix `world→plants`, where h and w are the values stored in `world→world_h` and `world→world_w`. Furthermore, it sets all of the matrix's entries to zero so that the world starts out with no food.

Add your implementation of `initialize_plants()` to *evolution.c*.

Listing 17.14: An outline of the function `add_plants()` in the file *evolution.c*.

```
1  static void add_plants(struct world *world)
2  {
3      int i = random(world→world_h);
4      int j = random(world→world_w);
5      world→plants[i][j]++;
6
7      if (world→eden_h > 0 &&  world→eden_w > 0) {
8          i = ...    ◀
9          j = ...    ◀
10         world→plants[i][j]++;
11     }
12 }
```

17.11.8 ▪ The function `add_plants()`

The function `add_plants()` that appears on line 21 in Listing 17.12 on page 177 is executed once at the beginning of every time-step. It is responsible for depositing one morsel of food in a random cell in the world and another morsel of food in a random cell in the Eden, if an Eden exists.

Listing 17.14 shows an outline. On lines 3 and 4 we pick random numbers i and j in the ranges $\{0,1,\ldots,h-1\}$ and $\{0,1,\ldots,w-1\}$, where h and w are the dimensions of the world, that is, `world→world_h` and `world→world_w`. Consequently, (i,j) represents a random cell in the world. Incrementing `world→plants[i][j]` on line 5 amounts to depositing a morsel of food there.

The purpose of the function's second half is to deposit an additional morsel of food in the Eden. You should be able to fill the excised parts with the help of the formula (17.2) on page 164.

17.11.9 ▪ The function `gene_to_activate()`

The function `gene_to_activate()` that appears on line 22 in Listing 17.12 on page 177 receives a chromosome, i.e., an array of eight genes, $\langle g_0, g_1, \ldots, g_7\rangle$, and selects a random gene to activate. It returns the index (i.e., a number in the set $\{0,1,\ldots,7\}$) of the selected gene.

If we were to select any of the genes with equal probability, then it would have been a matter of calling `random(8)` to get the desired index. But our task is a bit more complicated; we wish to assign higher probability of selection to genes of larger values. The way to do this is to consider eight line segments of lengths g_k for $k=0,1,\ldots,7$. Lay them back-to-back, and in order, from left to right, to obtain a line segment of overall length $L=\sum_{k=0}^{7} g_k$. Figure 17.3 shows this for the chromosome $\langle 2,4,1,1,6,3,1,2\rangle$, which has $L=20$.

Pick a random number r with uniform distribution in the range $\{0,1,\ldots,L-1\}$. (Letting `r = random(L)` will do that.) The key observation is that such a number is more likely to fall on a longer segment than a short one.

The outcome $r=9$, for instance, falls on the segment corresponding to the gene g_4, as seen in Figure 17.3. In that case `gene_to_activate()` will return 4 to signal that the gene g_4 is to be activated.

How does `gene_to_activate()` know that $r=9$ falls on the interval g_4? It sees that $r-(g_0+g_1+g_2+g_3)>0$ but $r-(g_0+g_1+g_2+g_3+g_4)<0$. That's how.

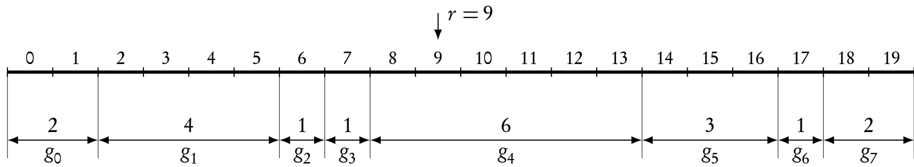


Figure 17.3: The chromosome $\langle 2,4,1,1,6,3,1,2\rangle$ is viewed as a sequence of line segments of lengths 2, 4, etc., encompassing an overall line segment of length 20. A random number r in the range $0 \le r < 20$ is more likely than not to fall on a longer line segment, thus selecting that segment for activation. The outcome $r = 9$ (or any $r \in \{8,9,10,11,12,13\}$ for that matter) selects the gene g_4, but only $r = 7$ selects the gene g_3.

Write a function `gene_to_activate()` that determines on which of the eight intervals the random number r falls and returns the index of that interval. Add it to your *evolution.c*.

17.11.10 ▪ The function `turn()`

The purpose of the function `turn()` that appears on line 23 in Listing 17.12 on page 177 is to turn an animal in place according to the description under the **Movement** paragraph on page 164. It receives a pointer to a **`struct`** `animal`, whence it determines the animal's current direction (it's in `animal→d`) and its chromosome (it's in `animal→genes`). It calls `gene_to_activate(animal→genes)` to determine the gene to activate. Let's say it's gene `k`. It turns the animal by incrementing `animal→d` by `k` modulo 8.

We may do that modular arithmetic in two equivalent ways. One way is through C's `%` operator: `a%b` gives the remainder of the division of the integer `a` by integer `b`.[62] Therefore we may set the animal's orientation after turning by

```
animal→d = (animal→d + k) % 8;
```

Alternatively, since we know that both `animal→d` and `k` are integers in the set $\{0,1,\dots,7\}$, then the animal's orientation after turning may also be set via

```
animal→d = animal→d + k;
if (animal→d > 7)
    animal→d -= 8;
```

Pick one or the other method, and finish writing the function `turn()`.

17.11.11 ▪ The function `move()`

The purpose of the function `move()` that appears on line 24 in Listing 17.12 on page 177 is to make an animal step forward (that is, in the direction it's facing) to the neighboring cell. The **Movement** paragraph on page 164 gives the specifics. Figure 17.1(b) on page 163 shows the neighboring cells in the directions $0,1,\dots,7$.

Suppose the animal is in the cell `(i,j)`. If it faces in the direction 0, after the move it will find itself in the cell `(i+1,j)`; see Figure 17.1(b). We see that the new position is `(i,j) + (1,0)`, where the addition is performed componentwise as in vectors. Similarly, if the animal is facing in the direction 1, then it will move to the cell

[62]Special care is called for when `a` or `b` are negative; consult your C reference book. That's not a concern here because we are dealing with nonnegative numbers.

Listing 17.15: The function `move()` in the file *evolution.c*. It makes an animal step forward (that is, in the direction it's facing) to the neighboring cell.

```
static void move(struct world *world, struct animal *animal)
{
    struct {
        int i;
        int j;
    } motion_vectors[] = {
        { 1 ,  0 },
        { 1 ,  1 },
        { 0 ,  1 },
        {-1 ,  1 },
        {-1 ,  0 },
        {-1 , -1 },
        { 0 , -1 },
        { 1 , -1 }};
    int i = animal→i + motion_vectors[animal→d].i;
    int j = animal→j + motion_vectors[animal→d].j;
    if (i < 0)
        i += world→world_h;
    else if (i ≥ world→world_h)
        i -= world→world_h;
    if (j < 0)
        j += world→world_w;
    else if (j ≥ world→world_w)
        j -= world→world_w;
    animal→i = i;
    animal→j = j;
}
```

`(i,j) + (1,1)`. Generally speaking, each direction `d` has an associated movement vector `(i',j')` such that the animal in the cell `(i,j)` and facing the direction `d` will move to cell `(i,j) + (i',j')`. It's convenient to introduce a structure such as

```
struct {
    int i;
    int j;
};
```

to hold integer pairs `(i',j')` and then make an array of length 8 of such structures to hold the `(i',j')` pairs for all eight possible directions. The declaration of the structure and the initialization of the array may be done all at once, as seen in Listing 17.15, which shows the function `move()` in its entirety. After computing the `(i,j)` index of the new cell on lines 15 and 16, we check whether it has gone outside the world rectangle, and if so, we bring it back—that's where the edges wrap around and the world gets its toroidal topology—by adding or subtracting the appropriate values.

17.11.12 ▪ The function `feed()`

The function `feed()` that appears on line 25 in Listing 17.12 on page 177 receives a pointer to an animal. It determines the animal's cell index `(i,j)` and then looks up the `(i,j)` entry of the `world→plants` matrix. If there is food there, it takes one morsel

and feeds it to the animal. The food count in the cell drops by one and the animal's energy increases by `world→plant_energy`. That's all. You should be able to do the details.

17.11.13 ▪ The function `clone()`

The function `clone()` that appears on line 26 in Listing 17.12 on page 177 receives a pointer to an animal. It creates an *identical* animal and returns a pointer to it. The function is quite straightforward. It calls `xmalloc()` as in

```
struct animal *new = xmalloc(sizeof *new);
```

to allocate memory for the clone and then copies all the data from the original to the clone. You should be able to do the details.

17.11.14 ▪ The function `mutate()`

The function `mutate()` that appears on line 27 in Listing 17.12 on page 177 receives a pointer to a chromosome and mutates it in the way described under the heading **Mutation** on page 165. Its implementation is quite straightforward, so I will leave the work to you, but here are two hints.

First, observe that `random(3)` returns a random integer in the set $\{0, 1, 2\}$; therefore `random(3)-1` is a random integer in the set $\{-1, 0, 1\}$. Second, you may have use for the preprocessor macro `MAX()` defined on line 9 in Listing 17.12, which finds the larger of its two arguments.

17.11.15 ▪ The function `reproduce()`

The function `reproduce()` that appears on line 28 in Listing 17.12 on page 177 receives a pointer to an animal and splits it in two. Specifically, it cuts down the animal's energy to half, calls `clone()` to make an exact copy, calls `mutate()` to mutate the cloned animal's chromosome, and then pushes the cloned animal into the linked list of the animals. Here is that function in its entirety:

```
1  static void reproduce(struct world *world, struct animal *animal)
2  {
3      animal→e /= 2;
4      struct animal *new = clone(animal);
5      mutate(new→genes);
6      world→herd = ll_push(world→herd, new);
7  }
```

Examine the code carefully, and especially note how `ll_push()` is used. When you feel comfortable with your understanding, insert the function in your *evolution.c*.

17.11.16 ▪ The function `update_world()`

The function `update_world()` that appears on line 30 in Listing 17.12 on page 177 updates the world by one time-step. It sees to it that (a) dead animals are removed from the linked list of animals; (b) each of the remaining animals turns and then moves to an adjacent cell, eats food if there is any, and reproduces if it has sufficient energy; and (c) adds a plant somewhere in the world, and another one somewhere in the Eden, if there is an Eden. Listing 17.16 shows my implementation. As always, go through it carefully and analyze every piece. Then insert it into your *evolution.c*.

Listing 17.16: The function `update_world()` in the file *evolution.c*.

```
1  static void update_world(struct world *world)
2  {
3      world→herd = remove_the_dead(world→herd);
4      for (conscell *p = world→herd; p ≠ NULL; p = p→next) {
5          struct animal *a = p→data;
6          turn(a);
7          move(world, a);
8          feed(world, a);
9          a→e--;
10         if (a→e ≥ world→reproduction_threshold)
11             reproduce(world, a);
12     }
13     add_plants(world);
14 }
```

17.11.17 ▪ The function `evolve()`

The function `evolve()` that appears on line 31 in Listing 17.12 on page 177 calls the function `update_world()` in a loop to update the world `n` times. It's quite a straightforward function, so I will let you write the whole thing, which is not much.

Evolution by its nature is an extremely slow process. To achieve any discernible effect, we need to run the simulation through millions of time-steps. Even on a fast computer this may take several minutes. Now, it is annoying to stare at a silent computer screen waiting for the program to finish and not knowing how much progress is being made. To give some indication of the program's state, it would be extremely nice if it could print the current step number once every 100,000 steps, say. Make your `evolve()` to do that.

17.11.18 ▪ The function `evolve_with_figs()`

The purpose of the function `evolve_with_figs()` that appears on line 32 in Listing 17.12 on page 177 is to take a sequence of snapshots of the state of the world. Specifically, if called with an argument of `n`, it produces `n` snapshots interleaved with `n-1` updates, as explained on page 167. Listing 17.17 shows an implementation. On line 5 we synthesize the name of the graphics file in the array `buf`. The sequence of file names will be *fig0000.eps*, *fig0001.eps*, and so on. On line 6 we pass the file name and the world structure to the function `world_to_eps()`, which extracts the necessary information and writes it as an EPS image into a *fig*.eps* file. Then on line 8 it calls `update_world()` to update the world by one time-step. The reason for the **`if`**-test is to avoid an extra update after the last snapshot. This is so that the *WDF* written by the program agrees with the last snapshot.

Add the function `evolve_with_figs()` to your *evolution.c*. This is the final addition to that file. You are done. Congratulations!

17.12 ▪ Experiments

Experiment #1: For the first experiment, try

```
$ ./evolution 1000 <world-no-eden.wdf >out-no-eden-1000.wdf
```

Listing 17.17: The function `evolve_with_figs()` in the file *evolution.c*.

```
1  static void evolve_with_figs(struct world *world, unsigned long int n)
2  {
3      char buf[16];      // an overkill – only 12 chars are needed
4      for (unsigned long int i = 0; i < n; i++) {
5          sprintf(buf, "fig%04lu.eps", i);
6          world_to_eps(world, buf);
7          if (i < n - 1)
8              update_world(world);
9      }
10 }
```

where *world-no-eden.wdf* is as in Section 17.3. Examine the contents of the output file *out-no-eden-1000.wdf*. You will find that the initial single animal has multiplied into 120 animals (this number may vary, depending on the random number generator that comes with your C library). Note that chromosomes of all the animals are very roughly the same as those of their original ancestors. There are no discernible evolutionary trends. One thousand time-steps are too few to evince evolutionary trends.

Experiment #2: Rerun the experiment over a greater time span:

```
$ ./evolution 10000 <world-no-eden.wdf >out-no-eden-10000.wdf
```

Examine the contents of the output file *out-no-eden-10000.wdf*. There are very slight indications that animals are developing a more dominant g_0 gene. That's the gene that promotes moving straight ahead.

Experiment #3: Rerun the experiment over a yet greater time span:

```
$ ./evolution 100000 <world-no-eden.wdf >out-no-eden-100000.wdf
```

Ah, now the trend is quite obvious; the evolution has led to dominant g_0, g_1, and g_7 genes. These are the genes that promote forward motion and sweeping vast expanses.

Experiment #4: Also try 1,000,000 and 10,000,000 time-steps:

```
$ ./evolution 1000000 <world-no-eden.wdf >out-no-eden-1000000.wdf
$ ./evolution 10000000 <world-no-eden.wdf >out-no-eden-10000000.wdf
```

What do you observe?

Experiment #5: Now let us try *world-and-eden.wdf*, which is just like *world-no-eden.wdf* but has a 10×10 Eden:

```
$ ./evolution 1000 <world-and-eden.wdf >out-with-eden-1000.wdf
```

Examine the output file *out-with-eden-1000.wdf*. There is no particularly interesting pattern there.

Experiment #6: Try

```
$ ./evolution 1000000 <world-and-eden.wdf >out-with-eden-1000000.wdf
```

Examine the output file *out-with-eden-1000000.wdf*. Do you see that there are two distinct species now? Recall that the program sorts the animals according to their distances away from the Eden.

Experiment #7: Also try

```
$ ./evolution 10000000 <world-and-eden.wdf >out-with-eden-10000000.wdf
```

The two distinct species stand out even more clearly now. Do you see the foreign species streaking through the Eden?

17.13 ▪ Animation

Try

```
$ ./evolution 100000 20 <world-and-eden.wdf >/dev/null
```

This runs the evolution for an initial 100,000 steps; then it takes 20 snapshots interleaved with updates. The snapshots are stored in the EPS files named *fig0000.eps* through *fig0019.eps*. Figure 17.2 on page 176 shows a sample snapshot.

The */dev/null* that appears in the command above is Unix's "black hole"; any stream directed to it just vanishes. I am redirecting the program's `stdout` stream to */dev/null* since I am not interested in seeing the *WDF* file that it writes—I have seen it already in the previous section's experiments.

How do we view the snapshots? There are many ways, but these depend very much on your computer's operating system. I can tell you what I use on Linux. You will have to find out what is available to you if your operating system is different.

- You may view the individual snapshots through any *PostScript* image viewer. The programs `display`, `evince`, and `gv` are among the many such utilities freely available on Linux platforms. I prefer `evince` for viewing single frames:

  ```
  $ evince fig0007.eps
  ```

- The `display` utility noted above is a part of the *ImageMagick* suite of image viewing and manipulation programs. The suite also contains the `animate` utility, which animates a sequence of images. Try

  ```
  $ animate -delay 25 fig*.eps
  ```

  You will have to give it a few seconds at startup since it has to read and digest all the frames ahead of time. Then it will show the 20 images in quick succession, as in a movie, and will loop forever. To stop the animation, move the mouse cursor inside the animation window and press the keyboard's "q" key.
  The `-delay 25` option makes the animation pause for 25/100 of a second between frames. Adjust it for slower or faster playback.

- I prefer more control in an animation. I like to stop it where I want to and move forward or backward one frame at a time. For that, see if you have the `feh` image viewing utility on your computer. It comes with most Linux distributions. You may also download the source for free from `<http://feh.finalrewind.org/>` and compile it yourself.
  `Feh` does not recognize *PostScript*. You will have to convert the *fig*.eps* files to a raster-based image format, such as PNG. This may be done through the `convert` utility that comes with *ImageMagick*:

  ```
  $ convert fig*.eps fig%04d.png
  ```

Within a few seconds, this will convert the *PostScript* files *fig0000.eps* through *fig0019.eps* to PNG images *fig0000.png* through *fig0019.png*. The `%04d` specifies 4-digit, zero-filled numbering of the output files. View the resulting images with

```
$ feh fig*.png
```

Press the keyboard's right and left arrows to move forward and backward through the images. Or just hold down one of those keys to let *feh* flip through those images as fast as it can, thus creating the illusion of an animation.

You will find a GIF animation in the book's website. I made that by packing a sequence of individual snapshots into an animated GIF file. Read the website to learn how it is done.

17.14 ▪ *Project Evolution*

Part 17.1. Complete and test the mini-project of Section 17.10.

Part 17.2. Run the experiments suggested in Sections 17.12 and 17.13. Then change the parameters in the *WDF*s and try again. Is evolution always successful?

Part 17.3. [optional] Entering large numbers such as 10000000 on the command-line is inconvenient because you have to carefully count the zeros. Modify the program's command-line parsing section to make it recognize the K, M, G suffixes[63] for numbers. Thus, 10K would mean ten thousand, 15M would mean 15 million, and 2G would mean 2 billion.

Part 17.4. [optional] The *WDF* defined in Section 17.3 carries no information about the distribution of food in the world. That's why our simulation begins with zero food. See if you can extend the definition of *WDF* to handle a prescribed food distribution. You may consider a syntax like

```
Food (i0,j0,n0) (i1,j1,n1) ...
```

to indicate that the cell (i_0, j_0) has n_0 morsels of food, etc. You will need to modify the functions `read_wdf()` and `write_wdf()` to handle such a specification.

This extension increases the program's versatility significantly since the *WDF* written by `write_wdf()` now contains the simulation's *complete* state upon the program's exit. That *WDF* then may be used as input to the program to continue the simulation where it was left off.

Part 17.5. [optional] Write a one-dimensional version of the evolution simulator. The world would be a line interval whose ends are identified; i.e., an animal exiting at one end emerges at the other end. (Topologically the world would be a circle.) The chromosome will consist of two genes $\langle g_0, g_1 \rangle$. If g_0 is activated, the animal will not turn. If g_1 is activated, it will do an about-face.

[63] These are the first letters of Kilo, Mega, and Giga, which correspond to multiplicative factors of 10^3, 10^6, and 10^9, respectively.

Chapter 18

The Nelder–Mead downhill simplex

Prerequisites: *Chapters 7, 8*

18.1 ▪ Introduction

The Nelder–Mead simplex method [49, 50] is an algorithm for finding the local minima of a function $f : \boldsymbol{R}^n \to \boldsymbol{R}$, called the *objective function* in optimization parlance. The idea behind it is very geometric. It starts with a suitably chosen initial simplex[64] in $\boldsymbol{R}^n$ and evolves it according to certain rules. The simplex undergoes iterative expansions, contractions, and displacements, and if everything goes according to plan, it moves "downhill" toward a local minimum of f. The term "downhill simplex" was used in [53] to refer to this behavior. A significant merit of the algorithm is that it is *gradient-free*, that is, it does not require a knowledge of the function's gradient, which may be difficult or expensive to compute. Consequently, it may be applied to minimize nondifferentiable functions, but see the caveats in Section 18.3.

18.2 ▪ The algorithm

The algorithm works in iterative cycles. I will explain what is done in one cycle. The cycles are repeated until certain convergence criteria are met. The calculations involve four parameters, $\beta_r, \beta_e, \beta_c$, and β_s, which permit some customization of the algorithm, although almost always these are taken as

$$\beta_r = 1, \quad \beta_e = 2, \quad \beta_c = \frac{1}{2}, \quad \beta_s = \frac{1}{2}.$$

Now, without further ado, let us begin with the description of the algorithm.

Rank the vertices. Let us write $\boldsymbol{x}^{(i)} \in \boldsymbol{R}^n$, $i = 0, 1, \dots, n$, for the simplex's $n+1$ vertices at the start of a cycle, and let $y_i = f(\boldsymbol{x}^{(i)})$ be the function values at those vertices. Three of the $n+1$ vertices play key roles in what happens next. The *best vertex* is

[64] A simplex in n dimensions is the generalization of a triangle (two dimensions) and a tetrahedron (three dimensions). It is the convex hull of $n+1$ points in the n-dimensional Euclidean space. Equivalently, it may be thought of as the "solid" object formed by a set of n vectors emanating from a common point in the n-dimensional space. If the vectors form a linearly independent set, then the simplex has a positive n-dimensional volume; otherwise it is a *degenerate simplex*.

one with the lowest function value (it's *best* because we are aiming to minimize the function). The *worst vertex* is one with the highest function value. The *next to worst vertex* is one with the highest function value after excluding the worst vertex. We write ia, iy, and iz for the indices of the best, next to worst, and worst vertices, respectively, because a, y, and z are the first, next to last, and last letters of the alphabet.[65]

The definitions of ia, iy, and iz are not quite unambiguous. What are the best, next to worst, and worst vertices if the vector of the function values is $[7,7,7,7]$ or $[5,5,1,1,7,7]$? We partially remove the ambiguity by requiring that ia, iy, and iz be *distinct*. Stated formally, we require that

$$\begin{array}{c} ia \neq iy, \quad iy \neq iz, \quad iz \neq ia, \\ y_{ia} = \min_i y_i, \quad y_{iz} = \max_i y_i, \quad y_{iy} = \max_{i \neq iz} y_i \end{array} \qquad \text{if } n \geq 2. \tag{18.1a}$$

Thus, if the function values are $[7,7,7,7]$, then $ia = 0$, $iz = 1$, $iy = 2$ will do, but $ia = iz = iy = 0$ is not acceptable. Similarly, if the function values are $[5,5,1,1,7,7]$, then ia will have to be one of 2 or 3; and either of the choices $iz = 4$, $iy = 5$ or $iz = 5$, $iy = 4$ will do.[66]

In defining the best, next to worst, and worst vertices above I have assumed implicitly that the simplex has at least three vertices, that is, the dimension of space $n \geq 2$. The one-dimensional case, $n = 1$, is special; the simplex is a line segment, and hence it has only two vertices. From the definition of iy it follows that $iy = ia$. Therefore picking distinct ia, iy, and iz is impossible. We make an exception in the case of $n = 1$ and change the first line of the conditions (18.1a) to

$$ia = iy \neq iz \qquad \text{if } n = 1. \tag{18.1b}$$

The ambiguities noted above are not the only ones. There are quite a few decisions in the rest of the algorithm that call for tie-breakings of various sorts, e.g., making comparisons with "$<$" versus "$\leq$". I will gloss over the precise tie-breaking rules—as did Nelder and Mead in their paper—therefore your program may not work exactly as someone else's that breaks the ties differently. See [40] for a set of precise tie-breaking rules that lead to a uniquely defined Nelder–Mead algorithm.

Reflect. Compute the centroid $\hat{\boldsymbol{x}}$ of the simplex face opposite to $\boldsymbol{x}^{(iz)}$,

$$\hat{\boldsymbol{x}} = \frac{1}{n} \sum_{i \neq iz} \boldsymbol{x}^{(i)}, \tag{18.2}$$

compute the "reflection" $\boldsymbol{x}^{(r)}$ of $\boldsymbol{x}^{(iz)}$ through $\hat{\boldsymbol{x}}$,

$$\boldsymbol{x}^{(r)} - \hat{\boldsymbol{x}} = -\beta_r (\boldsymbol{x}^{(iz)} - \hat{\boldsymbol{x}}), \tag{18.3a}$$

and let $y^{(r)} = f(\boldsymbol{x}^{(r)})$. If $\beta_r = 1$, as it almost always is, then $\boldsymbol{x}^{(r)}$ is the true reflection of $\boldsymbol{x}^{(iz)}$ through $\hat{\boldsymbol{x}}$ because then $\boldsymbol{x}^{(r)} - \hat{\boldsymbol{x}} = \hat{\boldsymbol{x}} - \boldsymbol{x}^{(iz)}$. The ***reflection*** diagram in Figure 18.1 illustrates this.

At this point the algorithm branches into four cases, depending on the relationship between the value of $y^{(r)}$ and the vertex values y_i, $i = 0, 1, \ldots, n$.

[65] Here I am breaking away from the tradition in mathematics of using single letters for variable names because I could not think of suitable replacements for the double-letter names ia, iy, and iz. From the programming point of view, however, these are quite natural; programmers routinely use multiletter symbols for variables.

[66] Here, and everywhere else in this book for that matter, array indices begin with zero.

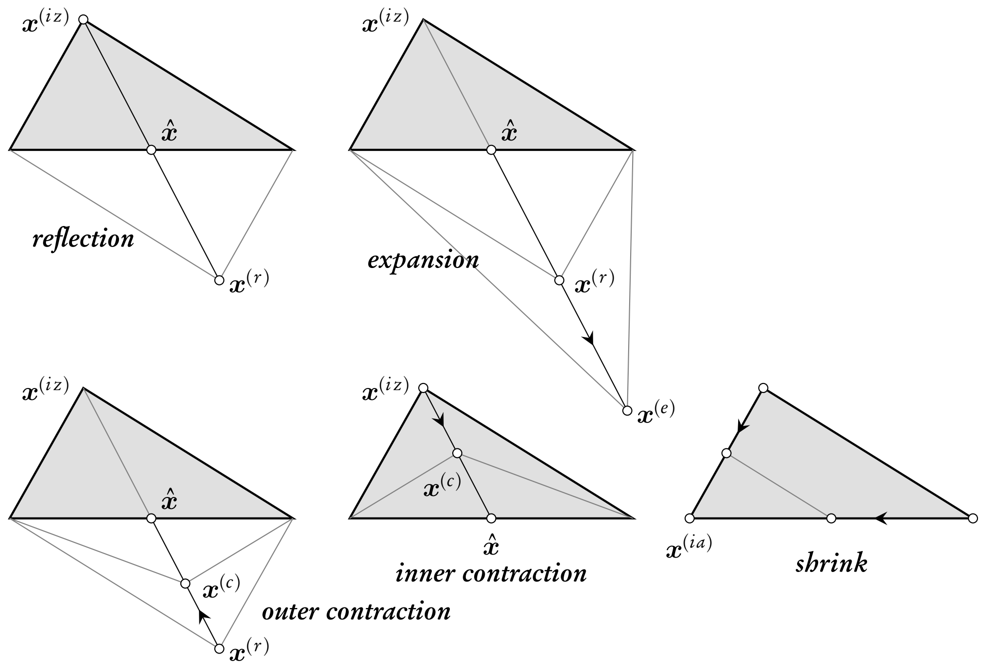


Figure 18.1: These diagrams show all possible transformations of a Nelder–Mead simplex in two dimensions ($n=2$) with the standard choices of the parameter values: $\beta_r = 1,\ \beta_e = 2,\ \beta_c = 1/2,\ \beta_s = 1/2$.

Case 1: $y^{(r)} < y_{ia}$

This is a terrific occurrence; what used to be the worst vertex changes to the best vertex upon reflection. Thus, the line that connects $\boldsymbol{x}^{(iz)}$ through $\hat{\boldsymbol{x}}$ to $\boldsymbol{x}^{(r)}$ appears to be a direction of steep descent of the function f. Take advantage of the opportunity, and expand the simplex in that direction by a factor of β_e. Thus, calculate a point $\boldsymbol{x}^{(e)}$ such that

$$\boldsymbol{x}^{(e)} - \hat{\boldsymbol{x}} = \beta_e(\boldsymbol{x}^{(r)} - \hat{\boldsymbol{x}}), \tag{18.3b}$$

and let $y^{(e)} = f(\boldsymbol{x}^{(e)})$. Usually one takes $\beta_e = 2$. This is illustrated in the ***expansion*** diagram in Figure 18.1. Replace the vertex $\boldsymbol{x}^{(iz)}$ with *the better of* $\boldsymbol{x}^{(e)}$ and $\boldsymbol{x}^{(r)}$, and end the iteration cycle.

Remark 18.1. This is slightly different from Nelder and Mead's original prescription which is the following: Pick $\boldsymbol{x}^{(e)}$ if $y^{(e)} < y_{ia}$; else pick $\boldsymbol{x}^{(r)}$.

Case 2: $y_{ia} \le y^{(r)} < y_{iy}$

Replace the vertex $\boldsymbol{x}^{(iz)}$ with $\boldsymbol{x}^{(r)}$, and end the iteration cycle.

Case 3: $y_{iy} \le y^{(r)} < y_{iz}$

Replacing $\boldsymbol{x}^{(iz)}$ by $\boldsymbol{x}^{(r)}$ under these conditions will constitute some improvement, but it will not be very useful since the worst vertex remains the worst vertex after the replacement. Thus, try another tack: push $\boldsymbol{x}^{(r)}$ toward the

centroid $\hat{\boldsymbol{x}}$ by a factor of β_c to arrive at a point $\boldsymbol{x}^{(c)}$,

$$\boldsymbol{x}^{(c)} - \hat{\boldsymbol{x}} = \beta_c(\boldsymbol{x}^{(r)} - \hat{\boldsymbol{x}}), \tag{18.3c}$$

and let $y^{(c)} = f(\boldsymbol{x}^{(c)})$. Usually one takes $\beta_c = 1/2$. The ***outer contraction*** diagram in Figure 18.1 illustrates this. If $y^{(c)} < y^{(r)}$, accept $\boldsymbol{x}^{(c)}$ as a replacement for the vertex $\boldsymbol{x}^{(iz)}$ and end the cycle; otherwise shrink the simplex (see below) and end the cycle.

Case 4: $y_{iz} \le y^{(r)}$

In this case a reflection is of no use at all since the worst point becomes even worse. Try another tack: push $\boldsymbol{x}^{(iz)}$ toward the centroid $\hat{\boldsymbol{x}}$ by a factor of β_c to arrive at a point $\boldsymbol{x}^{(c)}$,

$$\boldsymbol{x}^{(c)} - \hat{\boldsymbol{x}} = \beta_c(\boldsymbol{x}^{(iz)} - \hat{\boldsymbol{x}}), \tag{18.3d}$$

and let $y^{(c)} = f(\boldsymbol{x}^{(c)})$. The ***inner contraction*** diagram in Figure 18.1 illustrates this. If $y^{(c)} < y_{iz}$, accept $\boldsymbol{x}^{(c)}$ as a replacement for the vertex $\boldsymbol{x}^{(iz)}$ and end the cycle; otherwise shrink the simplex (see below) and end the cycle.

Shrink. The *shrink* process referred to in cases 3 and 4 above shrinks the simplex toward its best vertex $\boldsymbol{x}^{(ia)}$ by moving all other vertices toward $\boldsymbol{x}^{(ia)}$ by a factor of β_s:

$$\boldsymbol{x}^{(i)}_{\text{new}} - \boldsymbol{x}^{(ia)} = \beta_s(\boldsymbol{x}^{(i)} - \boldsymbol{x}^{(ia)}) \quad \text{for all } i \ne ia. \tag{18.3e}$$

Usually one takes $\beta_s = 1/2$. This is illustrated in the ***shrink*** diagram in Figure 18.1.

Are we there yet? There is no universally accepted stopping criterion for the Nelder–Mead algorithm. Nelder and Mead's original article [49] suggested the criterion

$$\sqrt{\frac{1}{n}\sum_{i=0}^{n}(y_i - \bar{y})^2} < \epsilon,$$

motivated by applications in statistics, where $\bar{y}$ is the average of the y_i's, and ϵ is a prescribed measure of accuracy. The stopping criterion proposed in [53] is

$$2\frac{|y_{iz} - y_{ia}|}{|y_{iz}| + |y_{ia}|} < \epsilon$$

for some prescribed ϵ. Either criterion is met if the variation of the objective function's values over the simplex's vertices is small. The simplex's size does not enter into the consideration.

Wright [80] quotes others who have suggested

$$\max_{i \ne ia} \|\boldsymbol{x}^{(i)} - \boldsymbol{x}^{(ia)}\| \le \epsilon \max(1, \|\boldsymbol{x}^{(ia)}\|)$$

for the stopping criterion, where ϵ is a prescribed measure of accuracy. This criterion is met if the simplex is sufficiently small. The objective function's values do not enter into the consideration.

The following hybrid criterion accounts for both the simplex's size and the objective function's values. It is quite robust and has worked well for my purposes, so that's what I prescribe for our program.

The user supplies a number, h, that establishes a "length scale" in $\boldsymbol{R}^n$. A good h should be of an order of magnitude commensurate with the features of interest in the objective function's domain. The length scale h serves dual purposes. For one thing, it enters the stopping criterion described below, and for another thing, it is used to set the simplex's initial size.

The user also supplies a dimensionless number, τ (for *tolerance*), that determines the algorithm's accuracy. The program lets $\epsilon = h \times \tau$ and declares that the algorithm has converged if

$$||\boldsymbol{x}^{(iz)} - \boldsymbol{x}^{(ia)}|| \leq \epsilon \quad \text{and} \quad |y_{iz} - y_{ia}| \leq \epsilon^2. \tag{18.4}$$

This assumes that ϵ is suitable for measuring the "smallness" of both $\boldsymbol{x}$ and y quantities. It also assumes that the objective function behaves like a quadratic at the minimizing point, which is generally the case if the function is smooth. You are free to replace these generic conditions with others which you deem better suited to problems of special interest to you.

To avoid runaway computations, should the algorithm fail to converge, the program keeps a count of the number of times it evaluates the objective function. It halts the iterations if the evaluation count exceeds a user-specified ceiling.

18.3 ▪ Problems with the Nelder–Mead algorithm

The Nelder–Mead algorithm is remarkable among computational algorithms in that there are hardly any convergence theorems about it. It is not known, for instance, whether the algorithm always converges for a function as simple as $f(x, y) = x^2 + y^2$.

McKinnon [45] has produced a clever example of a convex function f, defined on all of $\boldsymbol{R}^2$, and an initial simplex which, when evolved according to the Nelder–Mead algorithm, converges to a point which is *not* a local minimum of f. This rules out any unconditional convergence theorems, even for convex functions.

The function in McKinnon's example is three times continuously differentiable (i.e., it is in $C^3(\boldsymbol{R}^2)$ but not in $C^4(\boldsymbol{R}^2)$). It is not known whether additional smoothness may play a role in yielding a convergence theorem. The only concrete convergence result is in [40], where it is shown that the algorithm converges for a wide class of convex functions in $\boldsymbol{R}^1$.

Many modifications of the Nelder–Mead algorithm exist. Convergence theorems are available in some modified versions. See, e.g., Torczon [76], Kelley [30], and additional references in [40]. If you are interested in gradient-free optimization methods in general, you will find the survey article of Kolda, Lewis, and Torczon [36] quite informative.

In view of the lack of convergence theorems, and especially in view of McKinnon's counterexample, it may be a good idea to follow up an application of Nelder–Mead with an independent check to verify that the computed point is indeed a local minimum. One possibility, suggested in [53], is to restart the Nelder–Mead algorithm with a new initial simplex centered at the point returned by a previous run and verify that the simplex converges to the same point. Alternatively, one may use the minimizer produced by Nelder–Mead as an initial guess for a different algorithm, such as a conjugate gradient method. If the point produced by Nelder–Mead is indeed a minimum, any additional checks should incur only a small computational cost.

18.4 ▪ An overview of the program

Our implementation of the Nelder–Mead algorithm is in the file *nelder-mead.c*. The header file *nelder-mead.h* provides the application's interface. Additionally, the program relies on the previously developed `xmalloc()` for allocating memory and *array.h* for managing vectors and matrices. Therefore, following the recommendations of Chapters 2 and 6, the program's directory will look like this:

```
$ cd nelder-mead
$ ls -F
Makefile   demo-2D.c           nelder-mead.c          xmalloc.c@
array.h@   demo-constrained.c  nelder-mead.h          xmalloc.h@
demo-1D.c  demo-energy.c       rank-vertices-test.c
```

The *demo-*.c* files are drivers for various demonstrations. I will describe their contents and those of *rank-vertices-test.c* in the sections that follow.

Up to this point I have taken the objective function to be something of the type $f : \boldsymbol{R}^n \to \boldsymbol{R}$. This turns out to be too limiting in practice. Objective functions that occur in many interesting applications involve adjustable parameters. Therefore we expand our scope to include objective functions of the form $f(\boldsymbol{x}, \mu)$, where $\boldsymbol{x} \in \boldsymbol{R}^n$, and μ is a parameter. The C prototype of such a function is

```
double f(double *x, int n, void *params);     // the objective function
```

where `x` points to an array of length `n`. The third argument, being a *void pointer* (see Section 4.3), allows passing pointers to data of any type to such a function.

The program introduces a structure named **struct** `nelder_mead`, which is a repository of data that completely specifies a minimization problem. The minimization is performed by the function `nelder_mead()`, which has the prototype

```
int nelder_mead(struct nelder_mead *nm);
```

It receives the problem's statement in its `nm` argument, applies the Nelder–Mead algorithm as described earlier, and, if successful, inserts the computed minimum value and the coordinates of the minimizing point back into the structure. It returns the number of times that the objective function was evaluated in the process. That's useful information because it tells the user the amount of effort that was spent on finding the minimum. There is a second, and more important, reason for returning the evaluation count. The user specifies a number—we call it `maxevals` in the program—that sets an upper bound on the number of evaluations of the objective function. The iteration halts if the number of function evaluations exceeds `maxevals`. This prevents a runaway computation in case the algorithm fails to converge. Consequently, a representative call to the function `nelder_mead()` takes the form

```
nevals = nelder_mead(nm);
if (nevals < maxevals)
    printf("nelder-mead converged after %d evaluations\n", nevals);
else
    printf("no convergence after %d evaluations\n", nevals);
```

18.5 ▪ The interface

The header file *nelder-mead.h* is shown in Listing 18.1. It declares the prototype of the function `nelder_mead()` (on line 14) and the **struct** `nelder_mead` that holds the necessary data for communicating between the Nelder–Mead minimizer and application

Listing 18.1: The header file *nelder-mead.h*.

```
1  #ifndef H_NELDER_MEAD_H
2  #define H_NELDER_MEAD_H
3  struct nelder_mead {
4      double (*f)(double *x, int n, void *params);   // the objective function
5      int n;                        // dimension of the space
6      double **s;                   // (n+1) × n matrix: the simplex
7      double *x;                    // n-vector: the iteration's starting/ending point
8      double h;                     // the problem's length scale
9      double tol;                   // tolerance; stopping criterion
10     int maxevals;                 // max number of evaluations
11     double minval;                // the computed minimum value
12     void *params;                 // parameters to be passed to f()
13 };
14 int nelder_mead(struct nelder_mead *nm);
15 #endif /* H_NELDER_MEAD_H */
```

programs. Let us begin by examining the structure's details. Line numbers refer to those in Listing 18.1.

Lines 4 and 12: The member `f` of **`struct`** `nelder_mead` is a pointer to a user-supplied objective function. The function `nelder_mead()` will call `f()` repeatedly in the minimization process. When doing so, it will have to supply `f()` with a parameter in its third argument. It takes that parameter from the structure's `params` member (line 12). The user sets the value of the latter as appropriate when calling `nelder_mead()`.

Line 5: The member `n` is the dimension of the space over which the minimization takes place. That is, it is the n in $f : \boldsymbol{R}^n \to \boldsymbol{R}$.

Line 6: The member `s` is a pointer to an $(n+1) \times n$ matrix whose rows hold the coordinates of the simplex's vertices. Specifically, row i holds the coordinates of the vertex $\boldsymbol{x}^{(i)}$. As the simplex evolves, the entries of the matrix change. From now on the word *simplex* may refer to the geometric object or to the matrix `s`; they are isomorphic. The program offers two options regarding `s`:

Option 1: The user may set `s` to `NULL`, in which case the program will allocate memory for and construct a simplex matrix based on the initial point `x` (line 7) and the length scale `h` (line 8). It will free the memory allocated for that matrix before returning.

Option 2: The user may set `s` to a custom-built matrix (using `make_matrix()`) that specifies the initial simplex. The program will overwrite the matrix's entries as the simplex evolves. The user remains responsible for freeing the memory associated with the matrix.

Use the first option if you have no particular interest in building your own initial simplex. The second option is essential for constrained minimization; see Section 18.8.

Line 7: The member `x` points to a vector of length n and plays a dual role. Before calling `nelder_mead()` it holds a user-supplied initial point where the search is to begin. When `nelder_mead()` returns, it holds the coordinates of the minimizing point.

If you supply your own simplex matrix (see the previous paragraph), the program won't have a use for a starting point; therefore there is no need to initialize `x` in that case. Nevertheless, `x` must still point to a properly allocated n-vector because the program is going to write the coordinates of the minimizing point into it.

Line 8: The member `h` provides `nelder_mead()` with an idea of the problem's *length scale*. Without it, `nelder_mead()` cannot tell whether 1000 is small or large (small or large compared to what?). The value of `h` enters in two very different places in the program: one, in building the initial simplex; and two, in determining the stopping criterion.

To build the initial simplex (if the user has not supplied one) the program fills the rows of an $(n+1)\times n$ matrix with $n+1$ copies of the coordinates of the initial point `x` and then adds h to all members of the first subdiagonal:

$$s = \begin{pmatrix} x_0 & x_1 & x_2 & \cdots & x_{n-1} \\ x_0 & x_1 & x_2 & \cdots & x_{n-1} \\ x_0 & x_1 & x_2 & \cdots & x_{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_0 & x_1 & x_2 & \cdots & x_{n-1} \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 & \cdots & 0 \\ h & 0 & 0 & \cdots & 0 \\ 0 & h & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & h \end{pmatrix}. \tag{18.5}$$

Consequently, the rows of `s` correspond to the vertices of a simplex with one vertex at `x`, from which n legs of length h each emanate parallel to the coordinate axes.

Lines 9, 10, and 11: The members `tol` and `maxevals`, together with `h`, determine the calculation's stopping criteria. We assume that the iteration has converged if (18.4), or equivalently,

$$\|\boldsymbol{x}^{(iz)} - \boldsymbol{x}^{(ia)}\|^2 < \epsilon^2 \quad \text{and} \quad |y_{iz} - y_{ia}| \le \epsilon^2 \tag{18.6}$$

holds, where $\epsilon = h \times tol$. When this happens, `nelder_mead()` copies the value of the coordinates of the vertex $\boldsymbol{x}^{(ia)}$ into the vector `x`, sets the `minval` to the value of y_{ia}, and returns.

To prevent runaway computations, `nelder_mead()` keeps track of the number of times the objective function is evaluated. If the number of evaluations exceeds `maxevals`, it stops the iterations and returns.

Line 12: The user supplies a parameter here for `nelder_mead()` to pass as the third argument to the objective function `f()`. If the objective function has no use for parameters, pass `NULL` for its value.

18.6 ▪ The implementation

Listing 18.2 shows an outline of the implementation file *nelder-mead.c*. The only object there with external linkage is the function `nelder_mead()`; everything else is for internal use and therefore declared **`static`**. I will describe the purposes of the various functions in the following subsections.

18.6.1 ▪ The function `rank_vertices()`

Let us recall from Section 18.2 that $\{y_i\}_{i=0}^{n}$ are the objective function's values at the simplex's vertices, and the indices of the best, next to best, and worst vertices are ia, iy,

Listing 18.2: An outline of the file *nelder-mead.c*. Flesh out the parts marked with ▶.

```
1  #include <stdio.h>
2  #include "array.h"
3  #include "nelder-mead.h"
4  #define REFLECT     1.0
5  #define EXPAND      2.0
6  #define CONTRACT    0.5
7  #define SHRINK      0.5
8  ▶static inline void rank_vertices(double *y, int m,
9                              int *ia, int *iy, int *iz) ...
10 ▶static void get_centroid(double **s, int n, int iz,
11                              double *C) ...
12 ▶static inline void transform(double *P, double *Q, int n,
13                              double beta, double *R) ...
14 ▶static void shrink(double **s, int n, int ia) ...
15 ▶static inline void replace_row(double **s, int i,
16                              double **r) ...
17 ▶static int done(double **s, int n, double *y,
18                              int ia, int iz, double err2) ...
19 ▶int nelder_mead(struct nelder_mead *nm) ...
```

and iz. The function `rank_vertices()` that appears on line 8 in Listing 18.2 receives a pointer `y` to a vector of length `m` (where in fact `m` equals `n+1`, but the function need not know that) and calculates the three indices. Since a function in C cannot return more than one value, we return the three index values indirectly through pointers that the user provides as arguments. That's why the arguments ia, iy, and iz are declared as *pointers* to **`int`** in the function's prototype. I will leave it to you to write the function `rank_vertices()`. Have in mind the restrictions imposed in (18.1a) and (18.1b): The indices should be distinct (when $m \geq 3$) and $ia = iy \neq iz$ when $m = 2$. I suggest that you focus on the $m \geq 3$ case. If you do things right, your code will work without change with $m = 2$ as a bonus.

The function `rank_vertices()` works in the iteration's innermost loop; therefore we want it to be as efficient as possible. Toward that end, I have added the `inline` specifier on line 8 to help accelerate things a bit, if possible.[67] Implementing `rank_vertices()` is quite straightforward if one is willing to make two or three passes over the vector `y`. Doing it in a single pass is more challenging but not overly difficult. Challenge yourself to write `rank_vertices()` so that it works in one pass.

How do you know whether your `rank_vertices()` works correctly? I have written a stand-alone program called *rank-vertices-test.c*, which you can get from the book's website. Near the top of that file there is a place where you insert your implementation of the `rank_vertices()` function. Then compile and run the program:

```
$ cc -Wall -pedantic -std=c99 -O2 rank-vertices-test.c
$ ./a.out
```

If your `rank_vertices()` works correctly, the program will execute and exit silently; otherwise it will print diagnostics telling you what's not working.

Write your `rank_vertices()`, and be sure to test it with *rank-vertices-test.c*. If all is well, then insert it in *nelder-mead.c*.

[67] The `inline` specifier was introduced in C99. Remove it if you wish to retain C89 compatibility.

18.6.2 ▪ The function `get_centroid()`

The function `get_centroid()` that appears on line 10 in Listing 18.2 computes the centroid of the simplex's face opposite the vertex $\boldsymbol{x}^{(iz)}$ and writes its coordinates in the vector `C` that it receives as an argument. Equation (18.2) gives the formula for computing the centroid. The argument `s` points to the $(n+1)\times n$ simplex matrix (see Section 18.5).

I will leave the writing of that function to you. When you do so, it is likely that you will find that your code sweeps the matrix `s` vertically, down its columns. Computers tend to work more efficiently when they access memory in sequential order. The way we have constructed our matrices, that means in the row order. See if you can rewrite your code so that it sweeps the matrix `s` in the row order. This consideration may not be a significant issue with modern computers that have megabytes of CPU cache. It used to be a significant issue in the olden days. Nevertheless, it's a good exercise to see whether you can do it like the old pros did. See, however, the following remark.

Remark 18.2. The following code fragment sweeps the matrix `s[i][j]` in the column order:

```
for (j = 0; j < n; j++)
    for (i = 0; i < n + 1; i++)
        work with s[i][j]
```

A common misconception among the beginner programmers is to assume that swapping all occurrences of `i` to `j` as in

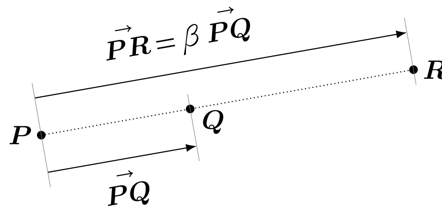
```
for (i = 0; i < n; i++)
    for (j = 0; j < n + 1; j++)
        work with s[j][i]
```

changes the sweep direction to the row order. But that's not true. Changing the names of the **for**-loop indices *does not* change the algorithm! The *names* of the index variables do not matter; for all we care, they could have been named `zork` and `pluto`.

To get things right, adhere strictly to the convention that `i` is the row index and `j` is the column index. Then build your algorithm around that convention.

18.6.3 ▪ The function `transform()`

Each of the geometric transformations (18.3a) through (18.3e) is of the form $\boldsymbol{R}-\boldsymbol{P}=\beta(\boldsymbol{Q}-\boldsymbol{P})$, where $\boldsymbol{P}$, $\boldsymbol{Q}$, and $\boldsymbol{R}$ are points in space. Letting $\vec{\boldsymbol{PQ}}$ and $\vec{\boldsymbol{PR}}$ be the vectors that extend from $\boldsymbol{P}$ to $\boldsymbol{Q}$ and $\boldsymbol{R}$, respectively, the equation above takes the form of a vector equation $\vec{\boldsymbol{PR}}=\beta\,\vec{\boldsymbol{PQ}}$ and has the geometric interpretation shown here for a $\beta>1$:



The transformation from $\vec{\boldsymbol{PQ}}$ to $\vec{\boldsymbol{PR}}$ is a stretch if $\beta>1$, a contraction if $0<\beta<1$, and a reversal if $\beta<0$. In particular, $\beta=-1$ produces a reflection because then $\vec{\boldsymbol{PR}}=-\vec{\boldsymbol{PQ}}$.

The function `transform()` that appears on line 12 in Listing 18.2 receives points $\boldsymbol{P}$, $\boldsymbol{Q}$ and a number β and computes $\boldsymbol{R}$ from $\boldsymbol{R}-\boldsymbol{P}=\beta(\boldsymbol{Q}-\boldsymbol{P})$, or equivalently,

$$\boldsymbol{R}=(1-\beta)\boldsymbol{P}+\beta\boldsymbol{Q}. \tag{18.7}$$

Write an implementation of the function `transform()`, and add it to your *nelder-mead.c*.

18.6.4 ▪ The function `shrink()`

The function `shrink()` that appears on line 14 in Listing 18.2 shrinks the simplex toward the vertex $\boldsymbol{x}^{(ia)}$ according to (18.3e). Actually (18.3e) is a set of n equations, each of which transforms one vertex of the simplex. The obvious way of implementing `shrink()` is to apply `transform()` in a loop.

Write an implementation of `shrink()`, and add it to your *nelder-mead.c*.

18.6.5 ▪ The function `replace_row()`

Consider the ***reflection*** diagram in Figure 18.1. What should the program do when the algorithm decides to accept $\boldsymbol{x}^{(r)}$ as the replacement for $\boldsymbol{x}^{(iz)}$? It will have to replace row `iz` of the matrix `s` with the coordinates of the point $\boldsymbol{x}^{(r)}$. Here is one possible way of doing this:

```
for (j = 0; j < n; j++)
    s[iz][j] = r[j];
```

(Here I have written `r` for the pointer to the array of length n that holds the coordinates of the point $\boldsymbol{x}^{(r)}$.) The same effect may be achieved with much greater efficiency by simply exchanging the pointers to the vector $\boldsymbol{x}^{(r)}$ and the matrix's row `iz`:

```
double *tmp = s[iz];
s[iz] = r;
r = tmp;
```

This accomplishes the previous version's **`for`**-loop (and n assignments) in exactly three assignments (independently of n). The savings can be significant if n is large. Since such exchanges are called for in several places in the algorithm, it makes sense to isolate the exchange code into a function. That is done in the function `replace_row()` that appears on line 15 in Listing 18.2. Here is a *very wrong* implementation of `replace_row()`:

```
/* wrong! */
static inline void replace_row(double **s, int i, double *r)
{
    double *tmp = s[iz];
    s[iz] = r;
    r = tmp;
}
```

That code's last line gives away what is wrong with it. Assigning a value to `r` just before leaving accomplishes *nothing* because `r` is a private variable within the function `replace_row()`; it goes away as soon as the function returns, so any assignment to it is a waste.

The correct way is to set things up so that `replace_row()` receives not the *value* of the pointer `r` but its *address*. Then `replace_row()` will be able to write into that

Listing 18.3: Here is the correct implementation of the function `replace_row()`. The argument `r` is the address of a pointer to a vector (in other words, a pointer to a pointer), and hence the double asterisks.

```
1 static void inline replace_row(double **s, int i, double **r)
2 {
3     double *tmp = s[i];
4     s[i] = *r;
5     *r = tmp;
6 }
```

address, which will retain its value after `replace_row()` returns. Listing 18.3 shows the correct implementation.

Be sure to understand what it does, and then add it to your *nelder-mead.c*.

18.6.6 ▪ The function `done()`

The purpose of the function `done()` that appears on line 17 in Listing 18.2 is to test whether the termination conditions in (18.6) have been reached. It returns *true* if yes and *false* if no. The arguments are the simplex `s`, the vector `y` of length `n+1` that holds the objective function's values evaluated at the vertices, the indices `ia` and `iz` of the best and worst vertices, and the argument `err2` which has the value $(h \times tol)^2$ because that's what is needed in (18.6).

Write an implementation of `done()`, and add it to your *nelder-mead.c*.

18.6.7 ▪ The function `nelder_mead()`

The prototype of the function `nelder_mead()` appears on line 19 in Listing 18.2. It implements the Nelder–Mead simplex algorithm with the help of the auxiliary functions already described.

The function is somewhat long; therefore I have broken it into two pieces. Listing 18.4 shows the top part of the function; it deals mainly with housekeeping tasks. It defines the aliases `s`, `n`, `h`, and `tol` as shorthands for `nm`→`s`, etc., because these occur so frequently in the code. The variable `err2` plays the role of ϵ^2 of (18.6).

The pointers `y`, `C`, `Pr`, `Pe`, `Pc` declared on line 8 are going to be set to point to various vectors. Specifically, `y` will point to a vector of length $n+1$ that will hold the values of the objective function at the simplex's vertices. The pointer `C` will point to vector of length n that will hold the coordinates of the centroid $\hat{\boldsymbol{x}}$ which we encountered in the description of the algorithm. The pointers `Pr`, `Pe`, `Pc` will point to vectors of length n each that will hold the coordinates of the points $\boldsymbol{x}^{(r)}$, $\boldsymbol{x}^{(e)}$, $\boldsymbol{x}^{(c)}$. The variables `yr`, `ye`, `yc` will hold the values of the objective function evaluated at the points $\boldsymbol{x}^{(r)}$, $\boldsymbol{x}^{(e)}$, $\boldsymbol{x}^{(c)}$. The variable `simplex_to_be_freed` is a Boolean flag; it is set to *false* if the user supplies the simplex matrix `s`; otherwise it is set to *true*. Near the end of `nelder_mead()`, just before it returns, the memory associated with the simplex matrix will be freed if `simplex_to_be_freed` is *true*. Otherwise the function's caller (who supplied the matrix) is responsible for freeing it.

The variable `fevalcount` keeps a count of the evaluations of the objective function. It is `nelder_mead()`'s return value.

The last five lines of Listing 18.4 call the macro `make_vector()` from *array.h* to allocate memory for the vectors `y`, `C`, `Pr`, `Pe`, `Pc`.

Listing 18.4: An outline of the function `nelder_mead()` – part 1.

```
1  int nelder_mead(struct nelder_mead *nm)
2  {
3      double **s = nm→s;
4      int n = nm→n;
5      double h = nm→h;
6      double tol = nm→tol;
7      double err2 = (h*tol)*(h*tol);
8      double *y, *C, *Pr, *Pe, *Pc;
9      double yr, ye, yc;
10     int ia, iy, iz;
11     int simplex_to_be_freed = 0;
12     int fevalcount;
13     int i, j;
14
15     make_vector(y, n+1);      // vertex values
16     make_vector(Pr, n);       // the reflected point x^(r)
17     make_vector(Pe, n);       // the expanded point x^(e)
18     make_vector(Pc, n);       // the contracted ponts x^(c)
19     make_vector(C, n);        // centroid of the face opposite vertex iz
```

Let us turn now to the second part of the function `nelder_mead()`, shown in Listing 18.5. Normally the user sets the member `s` of the **`struct`** `nelder_mead` to `NULL`, which has the effect of delegating the construction of the simplex matrix to the function `nelder_mead()`. In that case, the `make_matrix()` macro from *array.h* is invoked on line 21 to build that matrix. Then the flag `simplex_to_be_freed` is set to *true* to signal that the memory allocated here should be freed before the function returns. I have left out the part where the entries `s[i][j]` of the matrix are assigned values. Do it according to (18.5).

Lines 25–27 evaluate the objective function at the simplex's vertices and store their values in the vector `y`. Since that takes $n + 1$ function evaluations, `fevalcount` is initialized to `n+1`. Remember to update `fevalcount` upon every future evaluation of the objective function.

Let us turn for a moment to the last few lines of Listing 18.5. There we free all the allocated memory and return `fevalcount`, the number of times that the objective function was evaluated, to the caller. Note the checking of the `simplex_to_be_freed` flag; this function shouldn't free memory that was handed to it by the caller.

The bulk of Listing 18.5 is one big **`while`**-loop, beginning on line 29 and ending on line 57. It loops for as long as `fevalcount` $\leq$ `nm→maxevals`. The **`while`**-loop's body follows closely the algorithmic steps in Section 18.2. Thus, `rank_vertices()` is called on line 30 to determine the simplex's best, worst, and next to worst vertices; then `done()` is called to see whether the termination criteria are met, and if so, then `y[ia]`, which is the value of the objective function at the simplex's best vertex, is copied to `nm→minval`. This is what the caller will see as the sought minimum value. We also copy the coordinates of the simplex's best vertex, that is, row `ia` of the simplex matrix, into the vector `nm→x` to let the caller know where the minimum occurred. Line 33 is a placeholder for that. Do it yourself. The **`break`** on line 34 terminates the **`while`**-loop.

If the call to `done()` returns *false*, control transfers to line 36, where the process of reflecting, expanding, contracting, and shrinking the simplex begins. Thus, the centroid `C`

Listing 18.5: An outline of the function `nelder_mead()` – part 2.

```
20 if (s == NULL) {                          // build simplex if not supplied by user
21     make_matrix(s, n+1, n);
22     simplex_to_be_freed = 1;
23 ▶  assign values to s[i][j];             // see (18.5)
24 }
25 for (i = 0; i < n + 1; i++)
26     y[i] = nm→f(s[i], n, nm→params);
27 fevalcount = n+1;
28
29 while (fevalcount ≤ nm→maxevals) {
30     rank_vertices(y, n+1, &ia, &iy, &iz);
31     if (done(s, n, y, ia, iz, err2)) {
32         nm→minval = y[ia];
33 ▶      copy the best vertex into the vector nm→x ...
34         break;
35     }
36     get_centroid(s, n, iz, C);
37     transform(C, s[iz], n, -REFLECT, Pr);
38     yr = nm→f(Pr, n, nm→params);
39     fevalcount++;
40
41     if (yr < y[ia]) {                     // Case 1
42         transform(C, Pr, n, EXPAND, Pe);
43         ye = nm→f(Pe, n, nm→params);
44         fevalcount++;
45         if (ye < yr) {
46             replace_row(s, iz, &Pe);
47             y[iz] = ye;
48         } else {
49             replace_row(s, iz, &Pr);
50             y[iz] = yr;
51         }
52     } else if (yr < y[iy]) {              // Case 2
53 ▶      ...
54     } else {                              // Cases 3 and 4
55 ▶      ...
56     }
57 } // end of the while-loop
58
59 ▶  free vectors y, C, Pr, Pe, Pc
60 if (simplex_to_be_freed)
61     free_matrix(s);
62 return fevalcount;
63 }
```

of the face opposite the worst vertex is computed on line 36; then `transform()` is called to determine the reflection `Pr` of the worst vertex through `C`. Line 38 evaluates the objective function at `Pr` and assigns it to `yr`; then `fevalcount` is incremented. This brings the program to where the description of *the four cases* begins on page 195. On lines 41–51 I have implemented **Case 1**. I have left the remaining three cases to you to complete.

Remark 18.3. Remember that an application of `shrink()` in **Case 3** and **Case 4**, moves n of the simplex's $n+1$ vertices. Therefore you should update the vector `y[]` by reevaluating the objective function at the moved vertices and update `fevalcount` accordingly.

Remark 18.4. Once you implement the four cases, you will observe that the code for handling **Case 3** is almost identical to that of **Case 4**. See if you can streamline the logic to eliminate duplicate code.

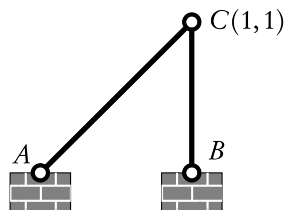
18.7 ▪ *Project Nelder–Mead*: Unconstrained optimization

Part 18.1. It's time to give our Nelder–Mead minimizer a try. The file *demo-2D.c* of Listing 18.6 gives a rather trivial demo of the utility. It is set up to minimize the function $f(x,y) = 3 + (x-2)^2 + (y-1)^2$. If all goes well, it will find the minimum value of 3 which occurs at the point $(2,1)$, or actually something very close to that, depending on the setting of the `tol` value. The demo illustrates the basics of initializing the `nelder_mead` structure, calling `nelder_mead()`, and interpreting the results. See if your code finds the minimum correctly. Here is what mine does:

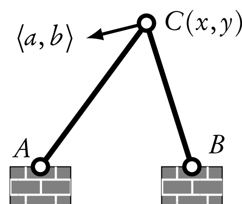
```
$ ./demo-2D
converged after 87 function evaluations
Computed solution: min = 3 at (1.99999, 1)
```

Remark 18.5. The initialization of **`struct`** `nelder_mead` in *demo-2D.c* takes advantage of a feature introduced in C99, whereby a structure's members may be initialized by naming them. See the discussion under the heading **Initializing structures** on page 6.

Part 18.2. Here is a more interesting minimization exercise (still in two dimensions). (This is an extremely special case of a *truss*—a subject treated fully in Chapter 19.) Consider a simple two-dimensional structure made of two elastic rods, AC and BC, hinged at their ends, where $A(0,0)$ and $B(1,0)$ are fixed, but $C(1,1)$ is free to move in the vertical plane when a load is applied to it. The rods elongate and shorten to accommodate the load, but they don't bend. We wish to determine the structure's deformed shape when a load vector $\langle a,b\rangle$ is applied at C.



The load-free state
C is at $(1,1)$



Deformation due to load $\langle a,b\rangle$
C moves from $(1,1)$ to (x,y)

The mechanics of an elastic rod are completely characterized by its *stored energy function* W that measures the energy required to stretch or compress a rod of unit length and

Listing 18.6: The file *demo-2D.c* is a very simple two-dimensional test for `nelder_mead()`.

```
#include <stdio.h>
#include "nelder-mead.h"
// f(x,y) = 3 + (x-2)^2 + (y-1)^2 = x^2 + y^2 - 4x - 2y + 8
static double obj_func(double *x, int n, void *params)
{
    return x[0]*x[0] + x[1]*x[1] - 4*x[0] - 2*x[1] + 8;
}
int main(void)
{
    double x[] = { 0.0, 0.0 };    // the initial point
    int evalcount;
    struct nelder_mead NM = {     // Alert! C99-style initialization!
        .f       = obj_func,      // the objective function
        .n       = 2,             // the dimension of the space
        .s       = NULL,          // delegate the construction of s
        .x       = x,             // initial point / final point
        .h       = 0.1,           // problem's scale
        .tol     = 1.0e-4,        // tolerance
        .maxevals = 1000,         // cap on function evaluations
        .params = NULL,           // no parameters
    };
    evalcount = nelder_mead(&NM);
    if (evalcount > NM.maxevals) {
        printf("No convergence after %d function evaluation\n",
                evalcount);
    } else {
        printf("converged after %d function evaluations\n",
                evalcount);
        printf("Computed solution: min = %g at (%g, %g)\n",
                NM.minval, x[0], x[1]);
    }
    return 0;
}
```

unit cross-sectional area made of that material. Specifically, changing that rod's length to λ requires an energy of $W(\lambda)$. The bar stretches when $\lambda > 1$ and shrinks when $0 < \lambda < 1$. A good choice for W is

$$W(\lambda) = \frac{\lambda^4}{24} + \frac{1}{12\lambda^2} - \frac{1}{8}. \tag{18.8}$$

Note that $W(1) = 0$ (no energy in the undeformed state), and $W(\lambda) \to \infty$ as either $\lambda \to \infty$ (large stretch) or $\lambda \to 0^+$ (severe compression). The energy required to stretch/compress a rod of cross-sectional area A from its natural length L to the length λL is $ALW(\lambda)$. I will take $A = 1$ in the rest of this chapter. See Chapter 19 for the full story.

Returning to the two-link structure described above, let us say the point C moves from $(1,1)$ to (x,y) when a load $\langle a,b \rangle$ is applied to it. The lengths of the links AC and BC change from the original $\sqrt{2}$ and 1 to $\sqrt{x^2+y^2}$ and $\sqrt{(x-1)^2+y^2}$, respectively.

It follows that structure's total energy is

$$\mathscr{E}(x,y) = \sqrt{2}W\left(\frac{\sqrt{x^2+y^2}}{\sqrt{2}}\right) + W\left(\sqrt{(x-1)^2+y^2}\right) - a(x-1) - b(y-1). \tag{18.9}$$

The last two terms express the work due to the displacement of the load.

It is shown in Chapter 19 that the structure's equilibria (in general there are multiple equilibrium states) correspond to the *stationary points* of the energy function $\mathscr{E}(x,y)$.[68] If the stationary point is a (local or global) minimum, the equilibrium is *stable*. If the stationary point is a (local or global) maximum or a saddle, the equilibrium is *unstable*.

Figure 18.2 shows the graphs and level sets of the energy function $\mathscr{E}(x,y)$ in two case studies. In Case study 1 the load is $\langle 0, -0.3 \rangle$. The energy function has three stationary points: D_1 is the global minimum; D_2 and D_3 are saddles. In Case study 2 the load is $\langle 0, -0.1 \rangle$. The energy function has five stationary points: D_1 and D_2 are minima; D_3, D_4, and D_5 are saddles. The drawings along the bottom of Figure 18.2 show the structure's three possible equilibrium configurations corresponding to Case study 1. Can you draw the five equilibrium configurations for Case study 2? You need not do any calculations for this; just examine the energy surface and the positions of the five equilibria.

The Nelder–Mead algorithm is ideally suited for finding the local minima of $\mathscr{E}(x,y)$.[69] Write a program *demo-energy.c* to compute the coordinates of the equilibrium point D_1 in Case study 1 and the equilibrium points D_1 and D_2 in Case study 2. In Case study 1 my program prints

```
$ ./demo-energy
Expected answer:   min = -0.6395281605 at (0.8319642234, -1.2505278260)
Computed solution: min = -0.6395281277 at (0.8315845906, -1.2505201990)
Converged after 71 function evaluations
```

The solution given on the "Expected answer" line is correct in *all its digits*. I computed that independently by solving the system of nonlinear equations $\{\partial\mathscr{E}/\partial x = 0, \partial\mathscr{E}/\partial y = 0\}$ with high precision in Maple and then pasted the answer into *demo-energy.c* for comparison with the Nelder–Mead result. As to the computation's parameters, I took `h=1.0`, `tol=1.0e-3` and chose $(-1,1)$ for the starting point just to be nasty since that's so obviously far from the expected equilibrium. I initialized the simplex to `NULL` to let the program build the simplex on its own.

To help you check your results, here are the coordinates and function values of the local minima D_1 and D_2 of Case study 2:

```
D1:  min = -0.2047379473 at (0.9208762303, -1.0926128103)
D2:  min = -0.0055676481 at (1.1554672673,  0.8776603419)
```

Remark 18.6. The load $\langle a, b \rangle$ enters the energy function (18.9) as a parameter. See if you can set up your code so that the energy function reads the load vector through its `params` argument.

Remark 18.7. The "smokestacks" in the graphs in Figure 18.2 are inherited from the vertical asymptote at zero of the function $W(x)$ in (18.8). They tell us that $\mathscr{E}(x,y) \to +\infty$ as the point C is pushed too close to the points A or B. (Do you see that?) In particular, $\mathscr{E}$ is undefined at A and B. This raises a serious concern: what would happen if a vertex

[68] A *stationary point* (also called a *critical point*) of a function $f(x) : \boldsymbol{R}^n \to \boldsymbol{R}$ is where $\nabla f(x) = 0$.

[69] Nelder–Mead cannot find saddle points. I found the coordinates of the saddle points D_2 and D_3 shown in Figure 18.2 by solving directly the nonlinear system of equations $\partial\mathscr{E}/\partial x = 0$ and $\partial\mathscr{E}/\partial y = 0$.

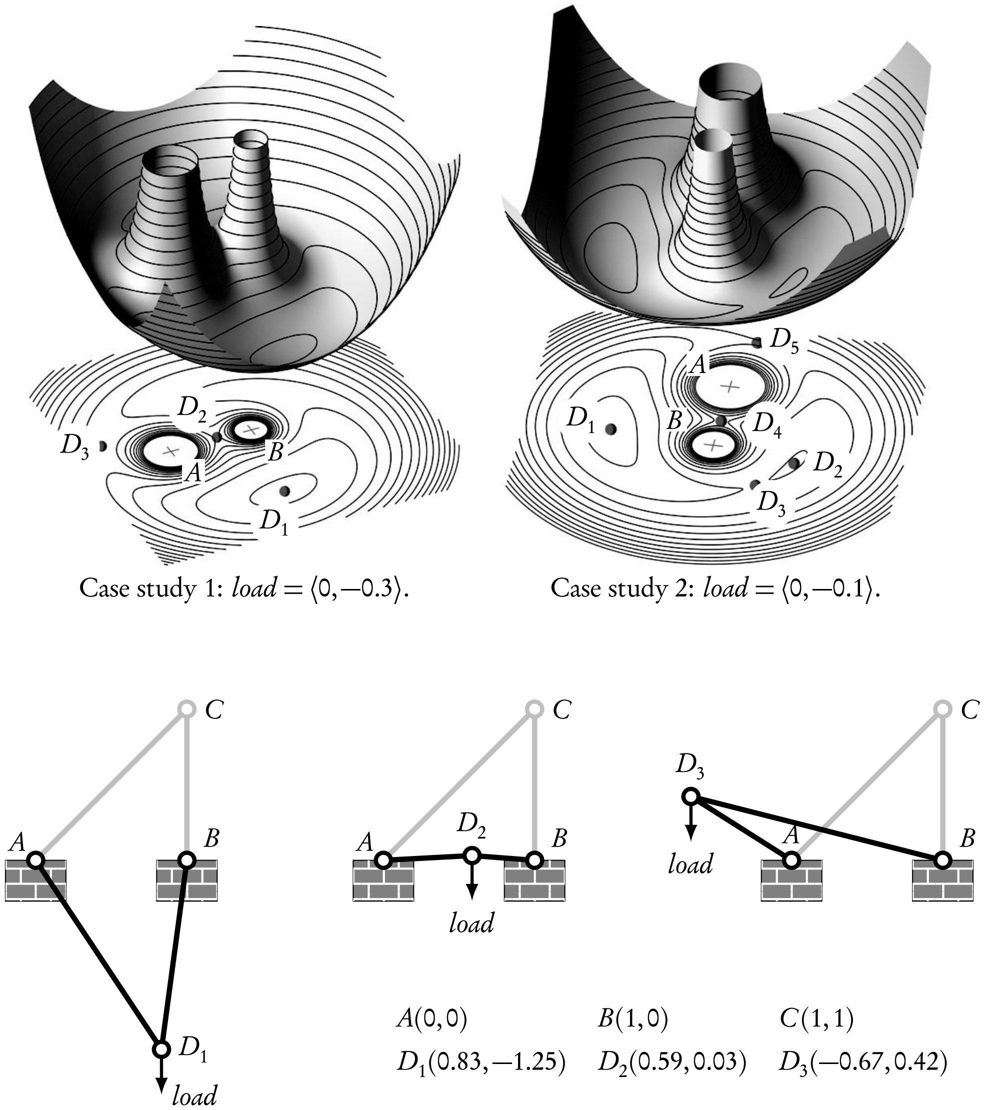


Figure 18.2: The plots at the top are the graphs and level sets of the energy function (18.9) corresponding to (a) Case study 1 (on the left) where the load is $\langle a, b \rangle = \langle 0, -0.3 \rangle$, and (b) Case study 2 (on the right) where the load is $\langle a, b \rangle = \langle 0, -0.1 \rangle$. The former has three stationary points: D_1 is the global minimum; D_2 and D_3 are saddles. The latter has five stationary points: D_1 and D_2 are minima; D_3, D_4, and D_5 are saddles. I have shown the plots of the energy surface from two different perspectives in order to convey a better sense of what they look like. The drawings along the bottom show the truss's three possible configurations corresponding to Case study 1. You do similar drawings for Case study 2.

of the simplex steps on either A or B during the iteration? The outcome of that event depends on your compiler. It may be set up to handle a floating point division by zero gracefully and produce an "infinity" which is larger than all possible floating numbers, or it may just crash the program. To be on the safe side, it would be a good idea to chop

off and flatten the smokestacks at a sufficiently large elevation to avoid them running to infinity. That certainly will not affect the locations of W's minima, which are the objects of our interest. Thus, noting that $W(10^{-3}) \approx 83333$, it would be safe to redefine W as

```
1 double W(double x)
2 {
3     double eps = 1.0e-3;
4     if (x < eps)
5         x = eps;
6     W(x) = ... ◄ // enter (18.8) here
7 }
```

Part 18.3. Try your Nelder–Mead on a function of one variable. Write a program *demo-1D.c* to minimize the function

$$f(x) = x^2 - 2x + 2 = (x-1)^2 + 1.$$

18.8 ▪ Constrained optimization

The simplex algorithm may be applied to minimization problems subject to constraints of certain types, as Nelder and Mead point out in their article [49]. Affine constraints are particularly easy to handle. Specifically, let $f : \boldsymbol{R}^n \to \boldsymbol{R}$ as before, and let $\mathscr{A}$ be an affine space in $\boldsymbol{R}^n$ defined by

$$\mathscr{A} = \{\boldsymbol{x} \in \boldsymbol{R}^n : A\boldsymbol{x} = \boldsymbol{b}\},$$

where A is an $m \times n$ matrix and $\boldsymbol{b}$ is an m-vector. We wish to compute

$$\min_{\boldsymbol{x} \in \mathscr{A}} f(\boldsymbol{x}). \tag{18.10}$$

We assume, without loss of generality, that $\text{rank}(A) = m$ because otherwise we may reduce the matrix to the row echelon form, discard the zero rows that appear at the bottom, and redefine A to what remains, without changing the space $\mathscr{A}$. We also assume $m < n$ because otherwise $\mathscr{A}$ reduces to a point or an empty set. With such an A, the constraint $A\boldsymbol{x} = \boldsymbol{b}$ is equivalent to a set of m scalar and linearly independent constraints:

$$\sum_{j=1}^{n} a_{ij} x_j = b_i, \qquad i = 1, \dots, m.$$

The key to extending the Nelder–Mead algorithm to the constrained problem (18.10) is the realization that the Nelder–Mead transformations, given in (18.3a) through (18.3e) and summarized in (18.7), are *affine-preserving* in the sense that if the points $\boldsymbol{P}$ and $\boldsymbol{Q}$ are in an affine space $\mathscr{A}$, then the point $\boldsymbol{R} = (1-\beta)\boldsymbol{P} + \beta\boldsymbol{Q}$ is also in $\mathscr{A}$. Thus, to solve the problem (18.10) it suffices to pick the initial simplex within $\mathscr{A}$. Then the transformed simplexes will remain within $\mathscr{A}$, and if the iteration arrives at a minimizer, it would be a solution of the constrained optimization problem (18.10).

One way to get the initial simplex to lie within $\mathscr{A}$ is to pick an unconstrained initial simplex in $\boldsymbol{R}^n$ and project it onto $\mathscr{A}$. It can be shown (see Section 18.10) that the operator P of the orthogonal projection onto $\mathscr{A}$ is given by

$$P\boldsymbol{x} = \left[I - A^T(AA^T)^{-1}A\right]\boldsymbol{x} + A^T(AA^T)^{-1}\boldsymbol{b}, \quad \boldsymbol{x} \in \boldsymbol{R}^n, \tag{18.11}$$

where I is the $n \times n$ identity matrix and the superscript T denotes the transpose.

Remark 18.8. If A is a $1 \times n$ matrix (that is, it is a row vector,) then the projection operator takes on a particularly simple form. For convenience, let us introduce the n-vector $\boldsymbol{a} = A^T$. Then the definition of the affine space $\mathscr{A}$ takes the form $\mathscr{A} = \{\boldsymbol{x} \in \boldsymbol{R}^n : \boldsymbol{a} \cdot \boldsymbol{x} = c\}$, where c is a constant.[70] It's not difficult to verify that the projection formula (18.11) collapses to

$$Px = x + \frac{1}{\|\boldsymbol{a}\|^2}\big(c - \boldsymbol{a} \cdot \boldsymbol{x}\big)\boldsymbol{a}. \tag{18.12}$$

Remark 18.9. A full-fledged application of either of the projection formulas (18.11) or (18.12) is an overkill in those simple cases where the nature of the projection is evident a priori. For instance, if $\mathscr{A}$ is a hyperplane perpendicular to one of the coordinates axes, as in

$$\mathscr{A} = \big\{\boldsymbol{x} = (x_1, x_2, \dots, x_7) \in \boldsymbol{R}^7 : x_5 = 12\big\}, \qquad (\mathscr{A} \text{ is perpendicular to the } x_5 \text{ axis}),$$

then projecting the simplex onto $\mathscr{A}$ amounts to setting all entries in column 5 of the initial simplex matrix to 12. Since the transformation (18.7) applied to the simplex matrix leaves a column of constants invariant, the simplex remains in $\mathscr{A}$ as it evolves.[71] This is how we enforce the immobility of a truss's fixed joints in Chapter 19.

Remark 18.10. Although projecting a simplex onto a proper affine space flattens it into a degenerate simplex of zero volume, this does not seem to have an adverse effect on the Nelder–Mead algorithm's operation.

Remark 18.11. It is possible to transform the constrained optimization problem (18.10) to an *unconstrained* one through elimination of variables. Specifically, we may solve the $m \times n$ linear system $A\boldsymbol{x} = \boldsymbol{b}$ for m of the unknowns in terms of the rest. (Recall that we are assuming that $\operatorname{rank}(A) = m$, and $m < n$.) We eliminate the solved variables in $f(\boldsymbol{x})$ and thus arrive at an *unconstrained* function $g : \boldsymbol{R}^{n-m} \to \boldsymbol{R}$. Applying the Nelder–Mead simplex algorithm to the unconstrained and lower-dimensional function g is more efficient compared to the projection method above—it requires fewer function evaluations, as experiments show. This has to be balanced against the programmer's time and effort in reducing f to g.

18.9 ▪ *Project Nelder–Mead*: Constrained optimization

Part 18.4. Minimize $f(x,y,z) = x^2 + y^2 + z^2$ subject to the constraint $x + y + z = 3$. It is easy to see that the minimum is 3 and occurs at $(1,1,1)$. Write a program, *demo-constrained.c*, to do that with the projection method. My program says

```
$ ./demo-constrained
converged after 140 function evaluations
min = 3 at (1, 1, 1)
```

For the parameters I took `h=0.1, tol=1.0e-5` and the initial point at $(1,2,3)$. My program builds the initial simplex according to (18.5) and then projects it onto the constraint space by applying (18.12).

[70] Thus, $\mathscr{A}$ is a hyperplane in $\boldsymbol{R}^n$, with $\boldsymbol{a}$ as a normal vector.

[71] This is pointed out in Nelder and Mead's article [49].

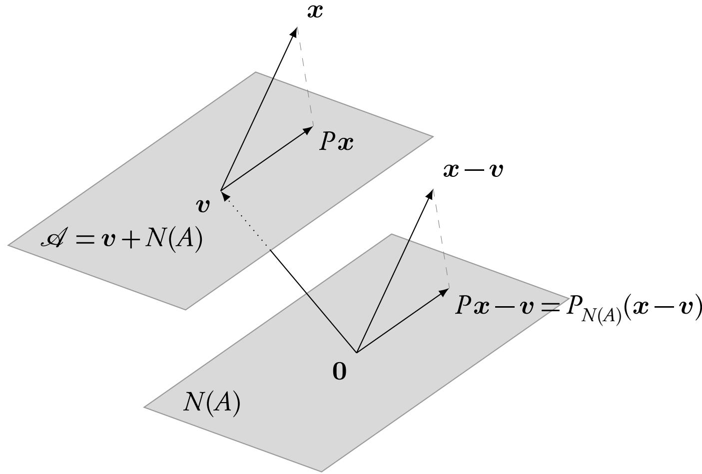


Figure 18.3: An imitation of a drawing on page 437 of [46].

18.10 ▪ Appendix: Orthogonal projection onto $Ax = b$

Although the projection formula (18.11) is well known among the specialists in linear algebra, it is difficult to locate in ordinary textbooks. It may be derived through an application of the *singular value decomposition theorem*, although that's an overkill since no singular values appear in the formula. Plesník [52] states the formula and refers the reader to Meyer's book [46]. I did not find an explicit form of the formula there, but the book offers all the ingredients needed for deriving it. Let me state the result formally as a theorem and then sketch a proof.

Theorem 18.1. *Consider the $m \times n$ matrix A and the m-vector $\boldsymbol{b}$, and let*

$$\mathscr{A} = \{x \in \boldsymbol{R}^n : Ax = \boldsymbol{b}\}.$$

Suppose $\operatorname{rank}(A) = m$. *Then the orthogonal projector $P : \boldsymbol{R}^n \to \mathscr{A}$ is given by*

$$Px = \big[I - A^T(AA^T)^{-1}A\big]x + A^T(AA^T)^{-1}\boldsymbol{b}, \quad x \in \boldsymbol{R}^n,$$

where I is the $n \times n$ identity matrix.

Proof. The affine space $\mathscr{A}$ is parallel to the nullspace $N(A)$ of A since for any u and v in $\mathscr{A}$ we have $Au = \boldsymbol{b}$ and $Av = \boldsymbol{b}$; therefore $A(u - v) = \mathbf{0}$. Figure 18.3 depicts the origin $\mathbf{0}$, the nullspace $N(A)$ of A, and the affine space $\mathscr{A}$, which is parallel to $N(A)$. Pick and fix a $v \in \mathscr{A}$, that is, $Av = \boldsymbol{b}$. It should be clear that $\mathscr{A} = v + N(A)$.

The diagram also shows an arbitrary point $x \in \boldsymbol{R}^n$ and its orthogonal projection Px onto $\mathscr{A}$. Subtracting v from both x and Px moves them to $x - v$ and $Px - v$, as shown, and thus the problem of projecting onto an affine space reduces to the problem of projecting onto a subspace. Writing $P_{N(A)}$ for the orthogonal projector onto $N(A)$, we have $Px - v = P_{N(A)}(x - v)$. Thus,

$$Px = v + P_{N(A)}(x - v).$$

It remains to characterize the projector $P_{N(A)}$. It can be shown (see page 434 of [46]) that if A is $m \times n$ and $\operatorname{rank}(A) = n$, then $P_{N(A^T)} = I - A(A^TA)^{-1}A^T$. It follows that if A is

$m \times n$ and $\operatorname{rank}(A) = m$, then $P_{N(A)} = I - A^T(AA^T)^{-1}A$. We conclude that

$$\begin{aligned} P\boldsymbol{x} &= \boldsymbol{v} + \left[I - A^T(AA^T)^{-1}A\right](\boldsymbol{x} - \boldsymbol{v}) \\ &= \boldsymbol{v} + \left[I - A^T(AA^T)^{-1}A\right]\boldsymbol{x} - \left[I - A^T(AA^T)^{-1}A\right]\boldsymbol{v} \\ &= \left[I - A^T(AA^T)^{-1}A\right]\boldsymbol{x} + A^T(AA^T)^{-1}A\boldsymbol{v} \\ &= \left[I - A^T(AA^T)^{-1}A\right]\boldsymbol{x} + A^T(AA^T)^{-1}\boldsymbol{b}. \end{aligned}$$

Chapter 19

Trusses

Prerequisites: *Chapters 7, 8, 9, 16, 18*

19.1 ▪ Introduction

A truss consists of a two- or three-dimensional network of slender *links* joined at *nodes*.[72] The junctions at the nodes are assumed to be hinge-like; that is, no bending moment is transmitted through a node. External loads are applied only at the nodes. The links are straight and do not bend.

A truss deforms under loads: links shrink and stretch, nodes move, internal and external reaction forces are developed, and the truss either finds a new equilibrium state or collapses. The natural (unloaded) geometry of a truss is called its *reference configuration*. The shape of a truss after it reaches equilibrium under a set of applied loads is called *deformed configuration*.

Trusses are ubiquitous in our daily lives. We see them in radio and cellular telephone transmission towers, electric power transmission lines, bridges, roof supports, construction crane booms, and orbital space station platforms, both in fiction and reality.

Figure 19.1 shows a very simple two-dimensional truss supported on two nodes. The left node is fully anchored; it cannot move, but the truss can pivot about it. The right node is on rollers; it is free to move horizontally but not vertically. Truss bridges are supported this way. Figure 19.2 is a photograph of a real truss bridge.

A major phase in the structural design and analysis of a truss is the determination of stresses and strains in its links induced by the externally applied loads. A well-designed truss utilizes its structural components in an optimal way; it is strong, lightweight, and economical. At the same time it should not deform excessively or break under the loads that it's meant to support.

In the traditional engineering practice, the geometry of a truss, i.e., the positions of its nodes and the links that connect them, is decided in advance. Then cross-sectional sizes and materials of the links are selected to enable the truss to sustain the anticipated loads. The geometric design is based on the engineer's intuition, his/her aesthetic considerations, the available structural materials, and his/her experience with robust structures that have withstood the test of time.[73]

---

[72] Links and nodes are also known as *members* and *joints*, respectively.

[73] Recent developments in the area of *topology design of structures* replace this ad hoc design process with sophisticated techniques that produce optimally designed structures without a priori assumptions on the geometry. See [8, 3] for more information.

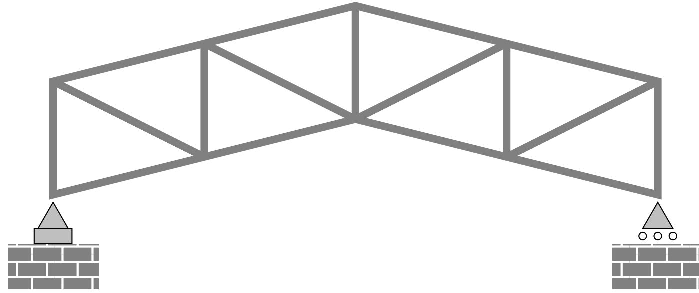


Figure 19.1: A simple roof truss. The left support is fixed. The right support is free to slide horizontally; it's a *roller support*.

Figure 19.2: The McKees Rocks Bridge in McKees Rocks, Allegheny County, Pennsylvania, built in 1931, is listed on the US National Register of Historic Places. *Photo courtesy of the Library of Congress, Prints & Photographs Division, HAER, Reproduction number HAER PA, 2-MCKRO,2–1. Photographer: Joseph Eliott, 1997.*

Trusses can be two-dimensional (also called flat or planar) structures, such as the modules used as roof supports, or three-dimensional, such as those used in power transmission towers. With the possible exception of trusses used as orbital platforms, a truss is anchored by a number of *supports* to an immobile foundation to prevent it from running away when forces are applied to it. In effect, supports constrain the possible movements of some of the nodes. In the truss shown in Figure 19.1, the leftmost node is completely immobilized, while the rightmost node is allowed to move horizontally but not vertically. Loads applied to any of the truss's nodes are transmitted via the links to the supports. The supports, in turn, exert reaction forces on the truss to achieve an equilibrium.

Solving a truss means computing the forces in the truss's links, and reaction forces at its supports, in response to a set of applied loads. The loads deform the truss. The deformation depends on the placement, magnitudes, and directions of the applied loads, as well as the mechanical properties of the materials that go into building the truss.

In structural mechanics, large deformations of structures such as bridges or booms of a crane are unacceptable. Such structures are designed purposely to deform only imperceptibly under loads within the design range; you wouldn't want to feel a bridge move underneath you as you drive over it. A consequence of the small deformation requirement is that for design purposes the shape of the bridge my be assumed to be independent of the applied forces. The effects of the infinitesimal extensions of the links have only second order effects on the shape and may be ignored in engineering practice. Solving such a truss reduces to solving a system of *linear equations*.

If, however, we admit large deformations whereby the geometry changes considerably under the applied loads, the equilibrium equations form a fully *nonlinear system*. Nonlinearity enters in the analysis of large deformations from two different sources: (i) *geometric nonlinearities* are introduced due to the finite deformation of the truss's geometry, and (ii) *mechanical nonlinearities* are introduced because the elastic response of materials under large deformation is generally nonlinear.

The program developed in this chapter solves trusses that may undergo arbitrarily large deformations. Admittedly, large deformations are of no great practical interest; however, they make for an interesting and fun academic exercise. I should emphasize, however, that this does not mean that the program developed here is worthless. Let's note that if the truss members are sufficiently stiff and the applied loads are not too large, then the truss's deformation will be small; therefore our solver will pick up the usual solution of an infinitesimally deformed truss.

Finally, let me point out that I have limited this chapter's programs to planar trusses only to simplify the exposition. The theory, however, is developed in the general n-dimensional context. Extending the programs to three-dimensional trusses is straightforward and poses no conceptual or programming challenges.

19.2 ▪ One-dimensional elasticity

An elastic solid is a material that deforms when you exert forces on it and returns to its original shape when you remove those forces. At least that's the intuitive idea. A rubber band, an automobile tire, a baseball, and a coiled spring are good examples of elastic solids. Less conspicuous elastic behavior is all around us. You sit on a seat cushion and it deforms to accommodate you; you rise and it bounces back to its original shape. A skyscraper sways in high winds and earthquakes. When the wind and seisms subside, it returns to its upright position. A bridge sags when a train passes over it. It resumes its normal shape when the train is gone. A paper clip is elastic in small deformations; it regains its original shape when you remove it from a thin stack of papers. It is not elastic under large deformations; if you bend it severely, it will not return to its original shape. What we have there is *plastic* behavior. That is the subject of study in the theoretical and experimental *plasticity*. We won't deal with plasticity in this book.

Theoretical elasticity is a deep and interesting subject. In this section I will present a *very special* case of elasticity as it applies to *one-dimensional deformations*, that is, deformations that are dominated by displacements in a single direction. The stretching of a rubber band is an example of one-dimensional deformations, but the deformation of an automobile tire where it hits the ground is not.

19.2.1 ▪ Stress

Figure 19.3 shows an elastic solid specimen in the shape of a cylinder of length L and cross-sectional area A in its natural (undeformed) state. It also shows the deformed shape

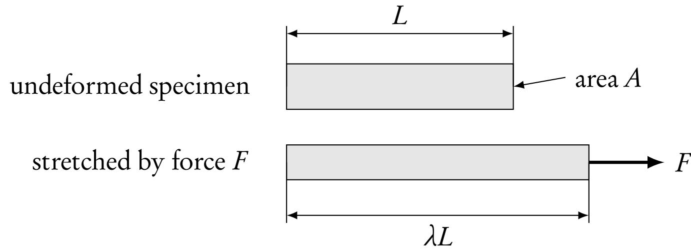


Figure 19.3: At the top: a cylindrical specimen of length L and cross-sectional area of A in its unstressed natural state. At the bottom: the specimen stretched to a length λL, $(\lambda > 1)$ due to the tensile force F applied to one end while the other end is held fixed. The cross-sectional area shrinks.

of the bar when it is pulled by a force F while one end is held fixed. Its length increases to λL $(\lambda > 1)$, and the cross-sectional area shrinks. The factor λ is called the bar's *stretch*.

By convention, tensile forces are regarded as positive and compressive forces as negative. Thus, stretching the bar corresponds to $\lambda > 1$ and $F > 0$. Compressing the bar corresponds to $0 < \lambda < 1$ and $F < 0$. It follows that the stretch-force function $F(\lambda)$ is such that $F(\lambda) > 0$ when $\lambda > 1$; $F(\lambda) < 0$ when $0 < \lambda < 1$; and $F(1) = 0$.

If the bar were twice as thick, that is, if its cross-sectional area were $2A$, it would take twice the force to stretch it by the same amount. In general, the force is proportional to the bar's cross-sectional area for a fixed amount of stretch. We express this as

$$F(\lambda) = A\sigma(\lambda). \tag{19.1}$$

The quantity $\sigma(\lambda)$, which measures force per unit cross-sectional area, is called the *stress* due to the stretch λ. The function σ is a mechanical property of the material much in the same way that density and electric resistance are. Note that the A in this definition is the bar's natural (undeformed) cross-sectional area, not the one after the deformation.[74]

The stretch-stress function $\sigma(\lambda)$ may be determined experimentally by subjecting a specimen to a series of tensile and compressive forces. The left diagram in Figure 19.4 shows the graph of the function σ of a hypothetical elastic material. We see that $\sigma(\lambda) \to -\infty$ as $\lambda \to 0^+$, indicating that it takes an infinite compressive force to squash the bar to zero length. We also see that $\sigma(\lambda) \to \infty$ as $\lambda \to \infty$, indicating that it takes an infinite tensile force to stretch the bar to infinite length.

Such idealized material is far from realistic. The experimental determination of the function in the $0 < \lambda < 1$ range is difficult, if not impossible, due to the unavoidable buckling that occurs under compression. Furthermore, a typical solid will either break or go into plastic behavior if severely compressed or stretched. Nevertheless, in our computations we will make the unwarranted assumption that $\sigma(\lambda)$ is defined for all $\lambda > 0$ since we don't intend to turn this book into a treatise on materials science.

[74]The corresponding concept in the three-dimensional nonlinear elasticity is called the *Piola–Kirchhoff stress*. It's also possible to define the stress by dividing the force by the area of the bar's current (deformed) cross-sectional area. That would be the *Cauchy stress*. In linear elasticity, which deals with infinitesimal deformations only, the two concepts coincide because the deformed and undeformed cross-sectional areas are only infinitesimally different.

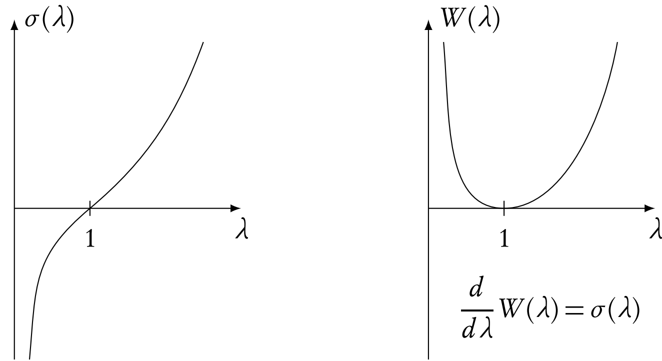


Figure 19.4: The graphs of the stress $\sigma(\lambda)$ (left) and the stored energy function $W(\lambda)$ (right) for a hypothetical elastic material. We have $\sigma(1)=0$ and $W(1)=0$ since stress and stored energy are zero in the undeformed state.

19.2.2 ▪ Energy

The *stored energy function* $W(\lambda)$ of an elastic material is defined by

$$W(\lambda)=\int_1^\lambda \sigma(\xi)\,d\xi, \tag{19.2}$$

or equivalently,

$$W'(\lambda)=\sigma(\lambda), \quad W(1)=0. \tag{19.3}$$

The right diagram in Figure 19.4 shows the graph of the energy function W corresponding to the previously discussed stretch-stress function σ.

To explain the significance of W, consider the gradual stretching of the bar from its original length L to a new length $\widehat{L}$, and let $\lambda=\widehat{L}/L$. At the intermediate stages of the process where the length of the bar is x ($L<x<\widehat{L}$), the stretch is x/L; therefore the pulling force is $A\sigma(x/L)$. The incremental work performed by the force as the length changes from x to $x+dx$ is $d\mathscr{W}=A\sigma(x/L)\,dx$. Therefore the total work performed in deforming the bar is

$$\mathscr{W}=\int_L^{\widehat{L}} A\sigma(x/L)\,dx \overset{\text{let } \xi=x/L}{=} \int_1^{\widehat{L}/L} LA\sigma(\xi)\,d\xi = LA\int_1^\lambda \sigma(\xi)\,d\xi = LA\,W(\lambda). \tag{19.4}$$

In the absence of a dissipative mechanism, the work is stored as internal elastic energy in the bar. Since LA is the volume of the (undeformed) bar, this says that the $W(\lambda)$ is the energy stored per unit volume due to the stretch λ. That explains the name *stored energy function* for $W(\lambda)$.

19.2.3 ▪ Small deformations

Structural engineering is concerned mainly with small deformations where the stretch λ is very close to 1. (You wouldn't expect a tall building to get noticeably shorter because of the weight of the snow on its roof.) In that sense, only the part of the graph of $\sigma(\lambda)$ near $\lambda=1$ is of interest. In that context, the relevant measure of deformation is the *strain*

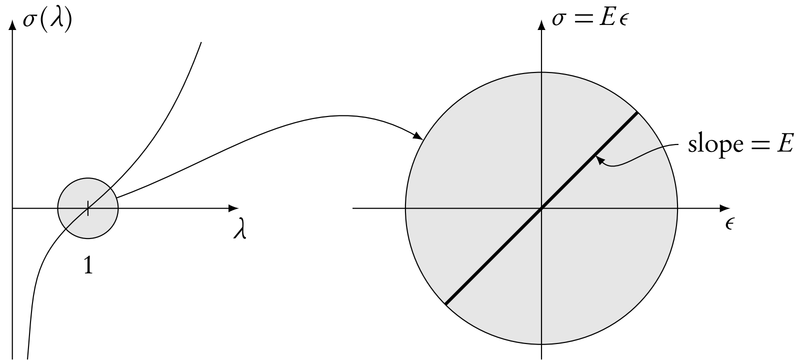


Figure 19.5: On the left is the graph of the stress-stretch function $\sigma(\lambda)$ of a hypothetical elastic material. Note that $\sigma(1) = 0$. The circle surrounding the point $\lambda = 1$ indicates the region of interest in practical structural engineering which is concerned mainly with $\lambda \approx 1$, or equivalently, $\epsilon = \lambda - 1 \approx 0$. The diagram on the right shows the enlarged image of that region. The stress-strain relationship is approximately linear. It's written $\sigma = E\epsilon$, where the slope $E = \sigma'(1)$ is called the material's *Young's modulus*.

(or more precisely, the *infinitesimal strain*) defined by $\epsilon = \lambda - 1$. In Figure 19.5, the graph on the left duplicates the previous graph of the stretch-stress function σ. The graph on the right magnifies the part of the graph near $\lambda = 1$ which now appears as a straight line. That line's slope, E, which equals $\sigma'(1)$, is called the material's *Young's modulus*. The equation of that line, $\sigma = E\epsilon$, is called the material's *linear strain-stress relationship*.

19.2.4 ▪ A concrete model

In our demonstration programs we need an explicitly defined stretch-stress function σ. Any function that has a graph like that of σ in Figures 19.4 or 19.5 will do. Its essential features are the following: (a) $\sigma(\lambda)$ is defined for all $\lambda > 0$, and (b)

$$\lim_{\lambda \to 0^+} \sigma(\lambda) = -\infty, \qquad \lim_{\lambda \to \infty} \sigma(\lambda) = \infty, \qquad \sigma(1) = 0, \qquad \sigma'(1) = E, \tag{19.5}$$

where the Young's modulus E is prescribed by the user. The function

$$\sigma(\lambda) = \frac{E}{6}\left(\lambda^3 - \frac{1}{\lambda^3}\right), \tag{19.6}$$

for instance, fits the bill, and that's what we will use in our program. That choice, however, is by no means unique. The functions

$$\sigma(\lambda) = \frac{E}{4}\left(\lambda^2 - \frac{1}{\lambda^2}\right) \qquad \text{and} \qquad \sigma(\lambda) = \frac{E}{3}\left(\lambda^2 - \frac{1}{\lambda}\right)$$

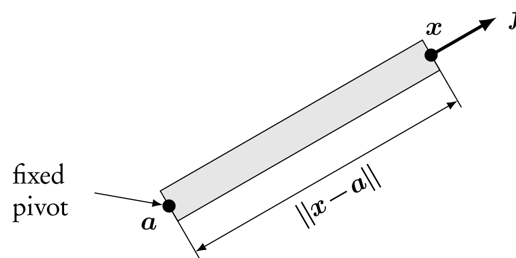
also share the same properties and will work just as fine.

For future reference, let us note that the stored energy function corresponding to the stress function (19.6) is

$$W(\lambda) = E\left(\frac{\lambda^4}{24} + \frac{1}{12\lambda^2} - \frac{1}{8}\right). \tag{19.7}$$

19.3 ▪ From energy to force

Consider a rod of natural length L and cross-sectional area A, one end of which is attached to, and can pivot about, a fixed point in space, call it $\boldsymbol{a}$, and the other end is free to be pushed, pulled, and moved about; call it $\boldsymbol{x}$.



Let $\boldsymbol{f}$ be the force vector applied at $\boldsymbol{x}$, and let $e(\boldsymbol{x})$ be the elastic energy stored in the rod. I will show that the gradient of $e(\boldsymbol{x})$ equals $\boldsymbol{f}$. Stated in symbols,

$$\nabla e(\boldsymbol{x}) = \boldsymbol{f}. \tag{19.8}$$

To see why, note that the rod's length is $\|\boldsymbol{x}-\boldsymbol{a}\|$; therefore, according to (19.4), its elastic energy is

$$e(\boldsymbol{x}) = LA\,W\left(\frac{\|\boldsymbol{x}-\boldsymbol{a}\|}{L}\right).$$

Compute the gradient by applying the chain rule from multivariable calculus. We get

$$\nabla e(\boldsymbol{x}) = LA\,W'\left(\frac{\|\boldsymbol{x}-\boldsymbol{a}\|}{L}\right)\frac{1}{L}\,\frac{\boldsymbol{x}-\boldsymbol{a}}{\|\boldsymbol{x}-\boldsymbol{a}\|}.$$

Now, $(\|\boldsymbol{x}-\boldsymbol{a}\|)/L$ is the bar's stretch, λ, and $W'(\lambda)=\sigma(\lambda)$ according to (19.3), so we get

$$\nabla e(\boldsymbol{x}) = A\sigma(\lambda)\frac{\boldsymbol{x}-\boldsymbol{a}}{\|\boldsymbol{x}-\boldsymbol{a}\|}.$$

According to (19.1), $A\sigma(\lambda)$ is the magnitude of the applied force. Since the expression $(\boldsymbol{x}-\boldsymbol{a})/\|\boldsymbol{x}-\boldsymbol{a}\|$ is a unit vector along the bar's direction, the product of the two is the applied force vector, as asserted.

Remark 19.1. Here I want to change the notation on you and rewrite (19.8) as

$$\frac{d}{d\boldsymbol{x}}e(\boldsymbol{x}) = \boldsymbol{f}. \tag{19.8'}$$

The change may look odd at first sight, but it's perfectly legal as long as one defines what is meant by the notation $d/d\boldsymbol{x}$, where $\boldsymbol{x}$ is not a scalar. But that's nothing new. Consider a function $f : X \to Y$, where X and Y are normed spaces (they need not be finite dimensional even). The *Fréchet derivative* Df of f at a point $x \in X$ is a bounded linear operator $(Df)(x) : X \to Y$ such that

$$\lim_{h\to 0}\frac{\left\|f(x+h)-f(x)-\big[(Df)(x)\big](h)\right\|_Y}{\|h\|_X} = 0.$$

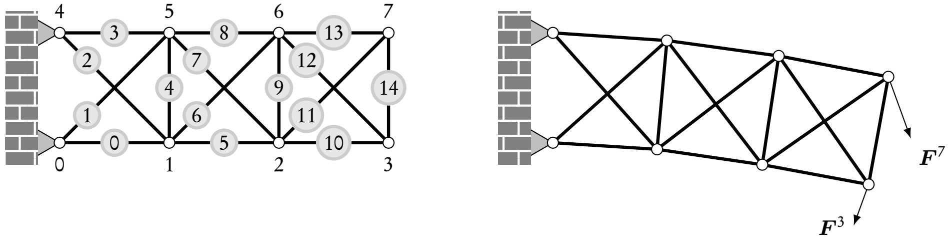


Figure 19.6: On the left we see a cantilever truss of eight nodes and 15 links in its natural (unloaded) state. Nodes and links are numbered independently, in no particular order. Nodes 0 and 4 are anchored and immobile; all others are movable. On the right we see the deformed truss under the loads applied at nodes 3 and 7.

When $X = \boldsymbol{R}$ and $Y = \boldsymbol{R}$, this agrees with the definition of the derivative in freshman calculus. If $X = \boldsymbol{R}^n$ and $Y = \boldsymbol{R}$, this agrees with the definition of the gradient in multivariable calculus. In (19.8') I have taken the liberty of changing the notation $(De)(\boldsymbol{x})$ to $\frac{d}{d\boldsymbol{x}}e(\boldsymbol{x})$, as it is more expressive for our purposes.

19.4 ▪ The energy of a truss

The left drawing in Figure 19.6 depicts a two-dimensional cantilever truss consisting of eight nodes (labeled 0–7) and 15 links (labeled 0–14). Nodes 0 and 4 are anchored to a wall and cannot move. The other nodes are free to move in response to applied loads. The right drawing in Figure 19.6 shows the truss's deformed state under loads applied to nodes 3 and 7. In general, loads may be applied to any of the nodes.

Generalizing from that concrete example, let us consider a two- or three-dimensional truss of n nodes, labeled $0, 1, \dots, n-1$, and l links, labeled $0, 1, \dots, l-1$. The order of the enumeration of the nodes and links is immaterial.

For each link p, where $p \in \{0, 1, \dots, l-1\}$, let L_p and A_p denote the link's undeformed length and cross-sectional area, W_p be the stored energy function of the link's material, and λ_p be the link's stretch after the truss is loaded.

Furthermore, for each node q, where $q \in \{0, 1, \dots, n-1\}$, let $\boldsymbol{F}^q$ denote the external load (a vector) applied to the node q, and let $\mathring{\boldsymbol{x}}^q$ and $\boldsymbol{x}^q$ be the node's position in space before and after loading, respectively.

Any arbitrary choice of positions $\{\boldsymbol{x}^0, \boldsymbol{x}^1, \dots, \boldsymbol{x}^{n-1}\}$ of the n nodes defines a (generally nonequilibrium) configuration of the truss in space. We define the *total energy* $\mathscr{E}$ associated with that configuration as

$$\mathscr{E}(\boldsymbol{x}^0, \boldsymbol{x}^1, \dots, \boldsymbol{x}^{n-1}) = \sum_{p=0}^{l-1} A_p L_p W_p(\lambda_p) - \sum_{q=0}^{n-1} \boldsymbol{F}^q \cdot (\boldsymbol{x}^q - \mathring{\boldsymbol{x}}^q). \tag{19.9}$$

The first summation expresses the total elastic energy stored in the links in that configuration. The second summation is the total work done by the loads due to their displacements. Let us observe that since the positions of a truss's nodes determines the (deformed) lengths of the truss's links, the stretches λ_p are easily computed in terms of $\{\boldsymbol{x}^0, \boldsymbol{x}^1, \dots, \boldsymbol{x}^{n-1}\}$.

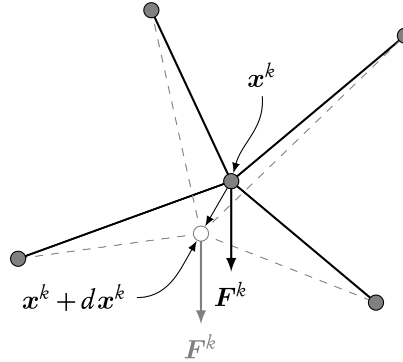


Figure 19.7: This shows a fragment of a truss, with a focus on the node x^k and four links that radiate from it. An external load F^k is acting on the node. To interpret the partial derivative $\partial\mathscr{E}/\partial x^k$, displace the node x^k by a small amount dx^k while keeping all other nodes fixed. We show that the change in energy in that deformation equals the sum of the internal and external forces applied to x^k.

We will see the significance of the definition (19.9) when we analyze its implications in the next section.

19.5 ▪ From energy to equilibrium

Consider the truss of n nodes and l links introduced in the previous section. Figure 19.7 shows a fragment of that truss, with a focus on some node x^k which has four links radiating from it. The choice of four links is for illustration only; the argument that follows is independent of the number of links that join at x^k. The nodes at the other ends of the links are shown as little circles, but their labels are not since those labels are irrelevant to the argument that follows.

The truss's total energy $\mathscr{E}(x^0, x^1, \dots, x^{n-1})$ is defined in (19.9). We may calculate the partial derivatives of $\mathscr{E}$ with respect to each of its arguments with the help of the differentiation formula (19.8'). But before we do that, let us examine the meaning of that operation. The differential $\big(\partial\mathscr{E}/\partial x^k\big)\cdot dx^k$ measures the change in $\mathscr{E}$ as the node x^k is displaced to a location $x^k + dx^k$ *while all other nodes are held fixed.* Figure 19.7 shows the "before" and "after" geometries.

Since all nodes except x^k are held fixed, the only links that change in length are those that connect to x^k. Therefore the summation over p in (19.9) reduces to a sum over those links that connect to x^k. Similarly, the summation over q reduces to a single term since nodes other than x^k do not move. In summary,

$$\frac{\partial\mathscr{E}}{\partial x^k}\cdot dx^k = \Big[\sum_{p\in \text{ links to } x^k} \frac{\partial}{\partial x^k}\big(A_p L_p W_p(\lambda_p)\big) - F^k\Big]\cdot dx^k.$$

According to the differentiation formula (19.8'), each term in the summation above equals the force vector f^p within the corresponding link. Therefore the equation above takes the form

$$\frac{\partial\mathscr{E}}{\partial x^k}\cdot dx^k = \Big[\Big(\sum_{p\in \text{ links to } x^k} f^p\Big) - F^k\Big]\cdot dx^k.$$

This is an extremely important result! The quantity in the square brackets is the vector sum of all forces (internal and external) applied to the point $\boldsymbol{x}^k$. When the truss is in equilibrium, *the sum of the forces applied to* $\boldsymbol{x}^k$ *should be zero* (this is Newton's second law; otherwise $\boldsymbol{x}^k$ must accelerate). We conclude that the left-hand side is zero. But since $d\boldsymbol{x}^k$ is arbitrary, then the derivative of $\mathcal{E}$ is zero. We state this as the following.

Theorem 19.1. *If a truss is in equilibrium with the external forces, then*

$$\frac{\partial \mathcal{E}}{\partial \boldsymbol{x}^k} = \mathbf{0}, \quad k = 0, 1, \dots, n-1.$$

In other words, the equilibrium configurations of a truss correspond to the stationary points of the function $\mathcal{E}$. There may be several stationary points of $\mathcal{E}$ corresponding to different states of equilibrium. The local minima of $\mathcal{E}$ correspond to stable equilibria. Figure 18.2 on page 210 shows that the energy function $\mathcal{E}$ can be quite complex even for a trivially simple truss; the "truss" there has only three nodes and two links!

Remark 19.2. Structural engineering deals mostly with infinitesimal strains. In that case the energy is a convex quadratic function of its arguments, and consequently it has a unique stationary point which is a local minimum. In that context, Theorem 19.1 is called the *minimum energy principle*.

The programs in the following sections find the equilibrium configurations of trusses by searching for the minima of $\mathcal{E}$ via the Nelder–Mead downhill simplex algorithm.

19.6 ▪ The *Truss Description File* (*TDF*)

A *Truss Description File* (*TDF*) is a plain text file that specifies a truss's geometry, materials, supports, and applied loads. I have supplied the sample *TDF* s *Cantilever.tdf* and *Pratt.tdf* in the book's website to get you started. You may create your own *TDF* s quite easily and add them to the collection.

To write a *TDF*, make a drawing of the intended truss, and then assign (a) coordinates (x_q, y_q) to every node, and (b) numeric labels (integers) to every node and every link. These may be assigned in no particular order, but there should be no duplicate labels within the nodes and no duplicate labels within the links. In the supplied *TDF* s the labels are assigned sequentially beginning with zero, but that's not a requirement; gaps within the sequence are quite alright. This allows you to insert or remove nodes and links anywhere within an existing *TDF*.

Listing 19.1 shows the contents of the file *Cantilever.tdf*. Figure 19.6 shows the truss's geometry, along with the node and link labels.

As you see in Listing 19.1, a *TDF* consists of three sections that deal with the nodes, the links, and the loads. Our truss solver utility reads the *TDF* through the `fetch_line()` function that we developed in Chapter 9. Therefore a *TDF* may be interspersed with comments and blanks since `fetch_line()` will trim them away. Let us examine the details of the *TDF*.

The *nodes* section. The nodes section begins with the string "begin nodes" and ends with "end nodes". In between are lines of the form

```
node  n:  x   y
```

where `n` is a node number, and `x` and `y` are that node's coordinates.

Listing 19.1: Here are the contents of the file *Cantilever.tdf*. The truss's geometry is shown in Figure 19.6. It has eight nodes (labeled 0–7) and 15 links (labeled 0–14), and loads are applied to nodes 3 and 7. Nodes 0 and 4 are fixed, as indicated by the asterisks.

```
# Cantilever.tdf
# A cantilever truss of eight nodes and 15 links

# Format: "node n: x y" where x and y are the node's coordinates
begin nodes
    node  0:   *0   *0
    node  1:    1    0
    node  2:    2    0
    node  3:    3    0
    node  4:   *0   *1
    node  5:    1    1
    node  6:    2    1
    node  7:    3    1
end nodes

# Format: "link n: n1 n2 A E" where n1 and n2 are the node
# numbers of the link's ends, A is the link's cross-sectional
# area, and E is the link's Young's modulus.
begin links
    link  0:    0  1  1  1
    link  1:    0  5  1  1
    link  2:    4  1  1  1
    link  3:    4  5  1  1
    link  4:    1  5  1  1
    link  5:    1  2  1  1
    link  6:    1  6  1  1
    link  7:    5  2  1  1
    link  8:    5  6  1  1
    link  9:    2  6  1  1
    link 10:    2  3  1  1
    link 11:    2  7  1  1
    link 12:    6  3  1  1
    link 13:    6  7  1  1
    link 14:    3  7  1  1
end links

# Format: "load n: fx fy" where (fx, fy) is the load applied to node n.
# Nodes not listed here have no externally applied loads.
begin loads
    load 7:      0.003     -0.010
    load 3:     -0.008     -0.005
end loads
```

An asterisk preceding a coordinate immobilizes it. Thus, nodes 0 and 4 in Listing 19.1 are immobile/fixed/anchored. A node description "`node n: x *y`" indicates that the node's `y` coordinate is fixed but `x` is free to move. Such a constraint is called a *roller support*. The right end of the truss in Figure 19.1 on page 216 is on a roller support.

The program recognizes four types of node specifications:

```
node  n:  x   y    # unconstrained
node  n: *x   y    # vertical roller
node  n:  x  *y    # horizontal roller
node  n: *x  *y    # anchored
```

Other types of constraints, such as nodes that roll on inclined planes, are possible, but our program does not implement those.

The *links* section. The links section begins with the string “begin links” and ends with “end links”. In between are lines of the form

```
links  n:  n1  n2  A  E
```

where `n` is a link number, and `n1` and `n2` are the node numbers of the link’s ends, `A` is its cross-sectional area, and `E` is its Young’s modulus. The program expects to be supplied with templates $\mathring{\sigma}(\lambda)$ for the stress function and $\mathring{W}(\lambda)$ for the stored energy function that satisfy the properties

$$\lim_{\lambda\to 0^+} \mathring{\sigma}(\lambda) = -\infty, \qquad \lim_{\lambda\to\infty} \mathring{\sigma}(\lambda) = \infty, \qquad \mathring{\sigma}(1) = 0, \qquad \mathring{\sigma}'(1) = 1,$$
$$\mathring{W}'(\lambda) = \mathring{\sigma}(\lambda), \quad \mathring{W}(1) = 0.$$

(Compare with (19.3) and (19.5).) The program takes $W(\lambda) = E\,\mathring{W}(\lambda)$ and $\sigma(\lambda) = E\mathring{\sigma}(\lambda)$ as the link’s stress function and stored energy function, where the Young’s modulus E is supplied by the user. In view of the models (19.6) and (19.7), the functions

$$\mathring{\sigma}(\lambda) = \frac{1}{6}\left(\lambda^3 - \frac{1}{\lambda^3}\right), \qquad \mathring{W}(\lambda) = \frac{\lambda^4}{24} + \frac{1}{12\lambda^2} - \frac{1}{8} \tag{19.10}$$

are suitable choices for such templates.

The *loads* section. The loads section begins with the string “begin loads” and ends with “end loads”. In between are lines of the form

```
load  n:  fx  fy
```

where `n` is a node number, and `fx` and `fy` are the horizontal and vertical components of the load vector applied to that node. There is no need to list every node in this section; nodes are assigned zero load by default.

19.7 ▪ An overview of the program

We are going to develop a *truss solver* utility. The solver reads the specifications of a truss from the `stdin`, computes its deformations under a set of given loads, and writes the result to the `stdout`. The shape of the deformed truss in Figure 19.6 on page 222 was computed that way.

The program relies on the *fetch-line* module of Chapter 9 to read input, the *xmalloc* module of Chapter 7) to allocate memory, the *Nelder–Mead* module of Chapter 18 to minimize the energy, the *array.h* header file of Chapter 8 to construct vectors and matrices, and the *linked lists* module of Chapter 16 to manage linked lists. Therefore, if you organize your files as suggested in Chapters 2 and 6, this project’s directory will look like this:

```
$ cd trusses
$ ls -F
Cantilever.tdf  fetch-line.h@   nelder-mead.h@  truss-to-eps.h
Makefile        interlude.c     truss-demo.c    truss.c
Pratt.tdf       linked-list.c@  truss-io.c      truss.h
array.h@        linked-list.h@  truss-io.h      xmalloc.c@
fetch-line.c@   nelder-mead.c@  truss-to-eps.c  xmalloc.h@
```

You will get the files *Cantilever.tdf*, *Pratt.tdf*, and *truss-to-eps.[ch]* from the book's website. The first two are sample *TDF* s (see Section 19.6). The last two provide a function `truss_to_eps()` that writes the geometry of a truss into a graphics file in the EPS format. I will explain the purposes and contents of the remaining files in the forthcoming sections.

The file *truss-demo.c* (details in Section 19.13) provides a demo of the truss solver. Running the compiled executable *truss-demo* as

```
$ ./truss-demo <Cantilever.tdf
```

reads the *TDF* *Cantilever.tdf*, solves the truss, i.e., determines the deformation under the given loads, and writes out the result in a format conforming to the *TDF* specification to the `stdout` (which is your computer screen by default). A typical output is several screenfuls long; therefore one often pipes the output into the Unix pager program "`less`", which displays the result one screenful at a time

```
$ ./truss-demo <Cantilever.tdf |less
```

or redirects it to a file, as in

```
$ ./truss-demo <Cantilever.tdf >Cantilever.out
```

See Chapter 3 if you need a refresher on Unix's stream redirection and pipes.

The program's output corresponding to the *TDF* *Cantilever.tdf* of Listing 19.1 is shown in Listing 19.2 and continues into Listing 19.3. The deformed configuration is shown in Figure 19.6 on page 222, and also in Figure 19.10 on page 249.

19.8 ▪ The interface

The header file *truss.h* in shown in Listing 19.4. It declares the prototype of the function `solve_truss()` (on line 31), the structures **`struct`** `node` and **`struct`** `link` that are capable of holding data for individual nodes and links, and the structure **`struct`** `truss`, which is a wrapper around the overall data that characterizes a truss.

The truss's nodes are stored in a linked list, and the `nodes_list` member of the **`struct`** `truss` (line 25, Listing 19.4) points to the head of that list. Similarly, the truss's links are stored in a linked list, and the `links_list` member of the **`struct`** `truss` (see line 26) points to the head of that list.

Let us examine the details of *truss.h*.

The `struct` `node` structure. The **`struct`** `node` structure, whose declaration begins on line 5, has members to hold the coordinates `(X,Y)` (uppercase `X` and `Y`) of a node in the truss's reference configuration and the coordinates `(x,y)` (lowercase `x` and `y`) of the same node in the deformed configuration. The members `xfixed` and `yfixed` are Boolean variables; `xfixed` is *true* if the x coordinate of the node is fixed and *false* otherwise. Similarly for `yfixed`. The `fx` and `fy` are the components of the load vector applied to this node. If the node is a support node, then `rx` and `ry` are the components

Listing 19.2: Here is the output of our truss solver corresponding to the *TDF Cantilever.tdf* of Listing 19.1. (The output continues into Listing 19.3.) The deformed configuration is seen in Figure 19.6 on page 222. The "`Math 625`" that appears near the top is the course number for which this book was written. Change it to something that's more relevant to you.
The placements of the asterisks in the `nodes` section appears to be quite different from those in *Cantilever.tdf*—there is excessive whitespace between an asterisk and the coordinate value that follows. Removing the excessive whitespace requires a certain amount of extra maneuvering which is not worth the effort since all extra whitespace is ignored by our *TDF* reader.

```
# Configuration of deformed truss computed by the
# Math 625 truss solver utility.

# node n: x y
begin nodes
node    0: *      0.0000 *      0.0000
node    1:        0.9530       -0.0636
node    2:        1.9136       -0.2008
node    3:        2.8828       -0.3804
node    4: *      0.0000 *      1.0000
node    5:        1.0420        0.9312
node    6:        2.0608        0.7885
node    7:        3.0567        0.6035
end nodes

# link n: n1 n2 A E
begin links
link    0:      0      1         1.00     1.00e+00
link    1:      0      5         1.00     1.00e+00
link    2:      4      1         1.00     1.00e+00
link    3:      4      5         1.00     1.00e+00
link    4:      1      5         1.00     1.00e+00
link    5:      1      2         1.00     1.00e+00
link    6:      1      6         1.00     1.00e+00
link    7:      5      2         1.00     1.00e+00
link    8:      5      6         1.00     1.00e+00
link    9:      2      6         1.00     1.00e+00
link   10:      2      3         1.00     1.00e+00
link   11:      2      7         1.00     1.00e+00
link   12:      6      3         1.00     1.00e+00
link   13:      6      7         1.00     1.00e+00
link   14:      3      7         1.00     1.00e+00
end links

# load nn: fx fy
begin loads
load    3:     -0.0080      -0.0050
load    7:      0.0030      -0.0100
end loads
```

Listing 19.3: This is a continuation of Listing 19.2.

```
# node n: rx ry
begin reactions
reaction    0:       0.0548       0.0049
reaction    4:      -0.0499       0.0101
end reactions

# link n: stress
begin stresses
stress    0:    -0.046042
stress    1:    -0.011899
stress    2:     0.009763
stress    3:     0.043440
stress    4:    -0.001223
stress    5:    -0.030186
stress    6:    -0.011875
stress    7:     0.010205
stress    8:     0.028330
stress    9:     0.000224
stress   10:    -0.014397
stress   11:    -0.011692
stress   12:     0.010383
stress   13:     0.012912
stress   14:    -0.000872
end stresses
```

of the support's reaction force, that is, the force that the support exerts on the truss. For nonsupport nodes, `rx` and `ry` will be zero.

The `struct link` structure. The *TDF* (see Section 19.6) declares the node *numbers* of a link's end points. The link structure, whose declaration begins on line 14 in Listing 19.4, does not store those numbers. Instead, it stores *pointers* to the corresponding node structures. This is much more useful since by following the pointer one gains access not only to the node number but also to complete data stored in that node structure. It is one of the program's tasks to convert the node numbers that it reads from the *TDF* to pointers to the corresponding nodes.

A link's natural (undeformed) length occurs in many places in the computation. Rather than computing it repeatedly on demand, we compute it once at the outset and store it as the member `L` in the link structure. The link structure also has members `A` and `E` to store the link's cross-sectional area and its material's Young's modulus, which are read from the *TDF* .

The `struct truss` structure. The **`struct`** `truss` structure declared on line 24 in Listing 19.4 is a wrapper around the overall data that characterizes a truss. The members `nodes_list` and `links_list` point to the heads of the linked lists of nodes and links, and thus they provide a complete specification of the truss's geometry and materials. Furthermore, a node structure contains, among other things, the components of the load vector applied to the node. Consequently, the function `solve_truss()` needs little other than a pointer to the **`struct`** `truss` to get its work done.

Listing 19.4: The header file *truss.h*.

```
1  #ifndef H_TRUSS_H
2  #define H_TRUSS_H
3  #include "linked-list.h"
4
5  struct node {
6      int n;                     // node number
7      double X, Y;               // node coordinates before deformation
8      double x, y;               // node coordinates after deformation
9      int xfixed, yfixed;        // true if coord is constrained
10     double fx, fy;             // components of the applied load
11     double rx, ry;             // components of reaction at support, if any
12 };
13
14 struct link {
15     int n;                     // link number
16     struct node *np1;          // pointer to node at one end
17     struct node *np2;          // pointer to node at the other end
18     double L;                  // link's natural length
19     double A;                  // link's cross-sectional area
20     double E;                  // the Young's modulus
21     double stress;             // link's stress
22 };
23
24 struct truss {
25     conscell *nodes_list;      // linked list of nodes
26     conscell *links_list;      // lined list of links
27     int nnodes;                // number of nodes
28     int nlinks;                // number of links
29 };
30
31 int solve_truss(struct truss *truss,
32                 double h, double tol, int maxevals);
33
34 #endif /* H_TRUSS_H */
```

The function `solve_truss()`**.** The function `solve_truss()` whose prototype appears on line 31 in Listing 19.4 receives a pointer to a truss structure and calculates its deformation by searching for a minimizer of the energy. It passes arguments `h`, `tol`, and `maxevals` to `nelder_mead()`, which actually performs the minimization. The configuration data of the deformed truss is printed to the `stdout` in a format compatible with a *TDF* . The function returns *true* upon success and *false* if anything goes wrong. Diagnostics are printed to the `stderr`.

19.9 ▪ Reading and writing: *truss-io.[ch]*

The task of reading and parsing a *TDF* is sufficiently complex to warrant isolating it into a separate file *truss-io.c* with an associated header file *truss-io.h*.

In fact, *truss-io.c* makes up the bulk of the program. In my implementation it is 332 lines long, while the solver itself (in file *truss.c*, more on that later) is only 177 lines long. Despite that length, the interface *truss-io.h*, shown in Listing 19.5, is quite simple.

Listing 19.5: The file *truss-io.h*.

```
1 #ifndef H_TRUSS_IO_H
2 #define H_TRUSS_IO_H
3 #include <stdio.h>
4 #include "truss.h"
5 struct truss *read_truss(FILE *stream);
6 void write_truss(struct truss *truss);
7 void free_truss(struct truss *truss);
8 #endif /* H_TRUSS_IO_H */
```

The function `read_truss()` reads a *TDF* from an input stream, aggregates the information into a truss structure, and returns a pointer to that structure. It allocates memory for the various structures as needed. Its sole argument, `stream`, is the stream from which it is to read the data. Our program invokes `read_truss()` with `stream` set to `stdin`, although other streams may be used equally well.

The function `write_truss()` acts almost as the inverse of `read_truss()`. It writes the data of the deformed truss to the `stdout` in a format consistent with the *TDF* specifications. It would be a trivial change to make it write to an arbitrary output stream rather than to the `stdout`, as done in `read_truss()`. You may do it that way if you wish.

The function `free_truss()` receives a pointer to a truss structure and frees the memory resources previously allocated by `read_truss()`. Normally you will call `free_truss()` at the end of the computation, just before exiting.

In addition to these three functions with external linkage, the file *truss-io.c* contains 10 functions with internal linkage, i.e., specified **`static`**. Listing 19.6 shows an outline of *truss-io.c*. I will describe its details in the following subsections.

19.9.1 ▪ The function `read_truss()`

The function `read_truss()` that appears on line 34 of Listing 19.6 is the controlling function of most of the rest of the file *truss-io.c*, and it has a big job to do. Nevertheless, it is quite clean and straightforward, as it delegates the messy work to the other functions. Its objective is to read a *TDF* from the specified stream, store the information in a **`struct`** `truss`, and return that structure's address. Listing 19.7 shows the contents of the function `read_truss()` in its entirety. Let's go through it line by line.

In the first few lines it declares a pointer to the truss structure that it's going to construct, as well as pointers to the linked lists of the truss's nodes and links. The variable `lineno` is initialized at `0`; it will be incremented upon reading the input stream's successive lines. Its purpose is to help the program issue helpful diagnostic messages, complete with line numbers, when problems occur in the input. There's not much use in saying "there is an error somewhere" without saying where.

After these initial preparations, the function commences to read, on line 8, the *TDF* 's node specifications section since that is the first section in a *TDF* . Actually, it delegates the task to the function `get_nodes()` which specializes in reading a *TDF* 's nodes. We will see the details of the implementation of `get_nodes()` in subsection 19.9.2. What we need to know at this stage is that upon success, `get_nodes()` constructs and returns a linked list of node structures that carries the information extracted from the *TDF* 's nodes section.

If `get_nodes()` fails for some reason, e.g., due to incomplete or garbled data, it frees whatever memory it has allocated in its aborted attempt and returns `NULL`. In that case

Listing 19.6: An outline of the file *truss-io.c*.

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <math.h>
4  #include "linked-list.h"
5  #include "fetch-line.h"
6  #include "xmalloc.h"
7  #include "truss.h"
8  #include "truss-io.h"
9  #define BUFLEN 128
10 #define SQ(x) ((x)*(x))
11
12 // functions for reading nodes
13  ▶ static void free_nodes_list(conscell *nodes) ...
14  ▶ static struct node *make_node(int n, double x, double y,
15         int xfixed, int yfixed) ...
16  ▶ static struct node *process_node_line(char *str) ...
17  ▶ static conscell *get_nodes(FILE *stream, int *lineno) ...
18
19 // functions for reading links
20  ▶ static void free_links_list(conscell *links) ...
21  ▶ static struct link *make_link(int n, int n1, int n2,
22         double A, double E, conscell *nodes_list) ...
23  ▶ static struct link *process_link_line(char *str,
24         conscell *nodes_list) ...
25  ▶ static conscell *get_links(FILE *stream, int *lineno,
26         conscell *nodes_list) ...
27
28 // functions for reading loads
29  ▶ static int process_load_line(char *str, conscell *nodes_list) ...
30  ▶ static int get_loads(FILE *stream, int *lineno,
31         conscell *nodes_list) ...
32
33 // functions with external linkage
34  ▶ struct truss *read_truss(FILE *stream) ...
35  ▶ void write_truss(struct truss *truss) ...
36  ▶ void free_truss(struct truss *truss) ...
```

the `if (...)` test on line 8 halts the function `read_truss()` and returns `NULL` to the caller.

If the reading of the nodes section is successful, control transfers to line 10, where another specialized function, `get_links()`, is called to read the *TDF*'s *links section*. If successful, `get_links()` constructs and returns a linked list of link structures that carries the information extracted from the *TDF*'s links section. Otherwise it frees whatever memory it has allocated in the process and returns `NULL`. In that case the `if (...)` test on line 10 halts the function `read_truss()`, but before returning `NULL` to the caller, it dutifully frees the memory associated with the linked list of nodes which it had obtained earlier.

The next segment, beginning on line 15, calls the function `get_loads()`, which specializes in reading a *TDF*'s *loads section*. Unlike the previous two segments, this function does not return a linked list. It simply reads the loads vectors from the *TDF* and inserts

Listing 19.7: The function `read_truss()` in the file *truss-io.c*. It reads a *TDF* from the `stream`, stores the information in a **`struct`** `truss`, and returns that structure's address.

```
1  struct truss *read_truss(FILE *stream)
2  {
3      struct truss *truss;
4      conscell *nodes_list;
5      conscell *links_list;
6      int lineno = 0;
7
8      if ((nodes_list = get_nodes(stream, &lineno)) == NULL)
9          return NULL;
10     if ((links_list = get_links(stream, &lineno, nodes_list))
11             == NULL) {
12         free_nodes_list(nodes_list);
13         return NULL;
14     }
15     if (!get_loads(stream, &lineno, nodes_list)) {
16         free_nodes_list(nodes_list);
17         free_links_list(links_list);
18         return NULL;
19     }
20
21     nodes_list = ll_reverse(nodes_list);
22     links_list = ll_reverse(links_list);
23
24     truss = xmalloc(sizeof *truss);
25     truss→nodes_list = nodes_list;
26     truss→links_list = links_list;
27     truss→nnodes = ll_length(nodes_list);
28     truss→nlinks = ll_length(links_list);
29     fprintf(stderr, "read %d nodes, %d links\n",
30             truss→nnodes, truss→nlinks);
31     return truss;
32 }
```

them into the corresponding nodes. That's the reason why it receives the linked list of nodes as an argument. If the reading is successful, it returns *true*; otherwise it returns *false*. In the latter case, the **`if`** `(...)` test on line 15 halts the function `read_truss()`, but before returning `NULL` to the caller, it dutifully frees the memory associated with the linked lists of nodes and links which had been constructed earlier.

If the reading of the *TDF* ends with success, control is transferred to line 21, where the function `ll_reverse()` from Chapter 16 is called to reverse the order of the nodes list. This is by no means necessary, but it's desirable: A linked list is assembled by pushing successive links into the head of the list, so it ends up in the reverse order of the arrival of the items. Without the reversal, the nodes in the program's output will be printed in the reverse order of the input. Line 22 serves a similar purpose in relation to the linked list of links.

At this point success is at hand. We call `xmalloc()` to allocate memory for a truss structure, hang the linked lists of nodes and links on it, print a informational message (to the `stderr`) announcing the number of nodes and links read, and return the address of the truss structure to the caller.

Listing 19.8: The function `get_nodes()` in the file *truss-io.c*. It is responsible for reading the nodes section of a *TDF*. The preprocessor macro `BUFLEN` is defined on line 9 in Listing 19.6 on page 232.

```
1  static conscell *get_nodes(FILE *stream, int *lineno)
2  {
3      conscell *nodes_list = NULL;          // an empty list
4      struct node *node;
5      char buf[BUFLEN];
6      char *str;
7      if ((str = fetch_line(buf, BUFLEN, stream, lineno)) == NULL) {
8          fprintf(stderr, "*** error: no node data in input\n");
9          return NULL;
10     }
11     if (strcmp(str, "begin nodes") != 0) {
12         fprintf(stderr, "*** error: no 'begin nodes' in input\n");
13         return NULL;
14     }
15     while ((str = fetch_line(buf, BUFLEN, stream, lineno)) != NULL) {
16         if (strcmp(str, "end nodes") == 0)
17             return nodes_list;
18         if ((node = process_node_line(str)) == NULL) {
19             fprintf(stderr, "*** error: bad syntax on input line %d\n",
20                     *lineno);
21             free_nodes_list(nodes_list);
22             return NULL;
23         }
24         nodes_list = ll_push(nodes_list, node);
25     }
26     fprintf(stderr, "*** error: no 'end nodes' in input\n");
27     free_nodes_list(nodes_list);
28     return NULL;
29 }
```

Remark 19.3. Normally the user will redirect the program's `stdout` to a file to capture the solution's data. That informational message noted above is not a part of the solution, so we print it to the `stderr`, which will flow to the screen and catch the user's attention, as explained in Chapter 3. The moral of the story: `stderr` is not exclusively for error messages.

19.9.2 ▪ The function `get_nodes()`

The function `get_nodes()` that appears on line 17 of Listing 19.6 on page 232 is responsible for reading the nodes section of a *TDF*. As we saw in subsection 19.9.1, it constructs and returns a linked list of node structures carrying the information extracted from the *TDF*.

In Listing 19.8 I have shown the function in its entirety, not because it's particularly long or complex, but because its gingerly, almost paranoiac, manner of stepping through the *TDF* sets the tone for the several similar functions which we will encounter.

It begins with defining an empty linked list called `nodes_list`, and it sets up a character buffer `buf[BUFLEN]`, which will hold successive lines read from the *TDF* through

calls to `fetch_line()`. The preprocessor macro `BUFLEN` is defined on line 9 in Listing 19.6 on page 232.

The first call to `fetch_line()` occurs on line 7. The expectation here is to find the string "begin nodes", which announces the beginning of the *TDF*'s nodes section; see Section 19.6 for the details of a *TDF*. If input ends before `fetch_line()` finds anything substantive—remember that `fetch_line()` skips over blanks and comments—`fetch_line()` returns `NULL`, in which case `get_nodes()` issues a diagnostic on line 8 and returns `NULL` to the caller.

On the other hand, if `fetch_line()` returns non-`NULL`, then control transfers to line 11, where the fetched string is compared against the target "begin nodes" with the help of the standard library's `strcmp()`. The latter returns zero if the stings exactly match, and nonzero otherwise.[75] If the comparison fails, `get_nodes()` issues a diagnostic on line 12 and returns `NULL` to the caller.[76]

If the string "begin nodes" is found, we expect that the subsequent lines will be either node specifications or the "end nodes" string. The **`while`**-loop that spans lines 15–25 repeatedly calls `fetch_line()` to extract, examine, and processes successive lines from the *TDF*.

Line 16 compares the fetched string against "end nodes". If it's a match, the nodes section has ended and the function's job is finished. The pointer `nodes_list`, which points to the head of the linked list of the nodes, is returned to the caller.

Otherwise, the helper function `process_node_line()` is called on line 18 to process the string. We will see the details of that function in the next subsection. What we need to know at the moment is that if the string is malformed, it rejects it and returns `NULL`, in which case a diagnostic message is printed (line 19), the linked list constructed thus far is freed, and `get_nodes()` returns `NULL`. If the string is well formed, however, it extracts the node number and the node's coordinates from the string, allocates memory for a node structure, inserts the extracted values in it, and returns that node's address. On line 24 the new node is pushed into the linked list of nodes, and then the **`while`**-loop repeats.

The normal return from the function `get_nodes()` is from line 17, which happens when the string "end nodes" is reached. If that string is missing, the **`while`**-loop falls off its end and the control transfers to line 26, where an appropriate diagnostic is printed, the linked list of nodes is freed, and `get_nodes()` returns `NULL`.

19.9.3 ▪ The function `process_node_line()`

The function `process_node_line()` appearing on line 16 of Listing 19.6 on page 232 receives a string as an argument which is expected to be a node specification line from a *TDF*. If the string meets the expectation, it allocates a **`struct`** `node`, fills in the node's members with data that it extracts from the string, and returns the node's address to the caller. If the string is malformed, it returns `NULL`.

The first task of `process_node_line()` is to decide which of the four types of node specifications listed on page 226 it has received. They are different only by the placement of the asterisks that indicate immobile coordinates. Thus, a string of the form "`node n: x *y`" specifies a node whose y coordinate is fixed but the x coordinate is

---

[75] Here we rely on the proper functioning of our `fetch_line()` function, which is designed to remove both leading and trailing spaces from a string. An errant whitespace here will foul up the comparison.

[76] Perhaps that diagnostic is too cryptic. It could be made more explicit by saying something like

```
fprintf(stderr, Expected 'begin nodes'but found %s on line %d\n",
        str, *lineno);
```

Listing 19.9: This fragment of the function `process_node_line()` (in the file *truss-io.c*) scans a string for one of the four possible input types listed on page 226. You will add code to handle the other three types. If the string fails to match one of the four types, the function returns `NULL`.

```
1  static struct node *process_node_line(char *str)
2  {
3      double x, y;
4      int n;
5      if (sscanf(str, "node %d: %lf %lf", &n, &x, &y) == 3)
6          return make_node(n, x, y, 0, 0);
7      else if (sscanf(str, "node %d: *%lf %lf", &n, &x, &y) == 3)
8      ...
9      else
10         return NULL;
11 }
```

Listing 19.10: The function `make_node()` in the file *truss-io.c*.

```
1  static struct node *make_node(int n, double x, double y,
2          int xfixed, int yfixed)
3  {
4      struct node *node = xmalloc(sizeof *node);
5      node→n = n;
6      node→X = node→x = x;
7      node→Y = node→y = y;
8      node→xfixed = xfixed;
9      node→yfixed = yfixed;
10     node→fx = node→fy = 0.0;
11     node→rx = node→ry = 0.0;
12     return node;
13 }
```

free to move (it's a horizontal roller support). Consequently, in the corresponding node structure (see Listing 19.4 on page 230) it should set `xfixed = 0` and `yfixed = 1`.

Our `process_node_line()` delegates the creation of the node structure and setting the values of its various members to yet another helper function, `make_node()`. We will examine the details of `make_node()` in the next subsection. What we need to know at the moment is that it receives the values of `n`, `x`, `y`, `xfixed`, and `yfixed` as arguments; then it allocates memory for a **`struct`** `node`, fills in those five values, and returns that node's address. Equipped with that knowledge, you should be able to complete the function `process_node_line()`, a fragment of which is shown in Listing 19.9.

19.9.4 ▪ The function `make_node()`

The function `make_node()` that appears on line 14 of Listing 19.6 on page 232 allocates memory for a node structure and fills the values of the `n`, `x`, `y`, `xfixed`, and `yfixed` members with those that it receives as arguments. Listing 19.10 shows its implementation. As you see, it does more than just filling in the five members.

For one thing, on lines 6 and 7 it sets the node's undeformed coordinates `(X,Y)`, as well as the deformed coordinates `(x,y)`, to the values received (which are the values

read from the *TDF*). In subsection 19.12.6 we will see that the truss solver constructs the Nelder–Mead initial simplex from the values stored in the (x,y) coordinates. Consequently, the search for the minimum energy begins from the truss's undeformed state. In Part 19.5 of this chapter's *Projects* section we will also see that the user has the opportunity to override this default choice if he/she so wishes.

For another thing, the load vector components `fx` and `fy` are set to zero as defaults. These may be overwritten later when the program reads the loads section of the *TDF*. If a node is not mentioned in that section, the default values remain in effect.

Finally, the support reaction force components, `rx` and `ry`, are set to zero as defaults. They will remain zero for nonsupport nodes. The reaction forces at the support nodes will be calculated and updated after the truss is solved.

Remark 19.4. This ends the chain of function that collectively read a *TDF*'s node description section. Each function delegates the details of a complex task to the next "specialist" function:

```
read_truss() ⇆ get_nodes() ⇆ process_node_line() ⇆ make_node()
```

Such a division of labor helps to make the logic of the program transparent and avoids excessively long and complex functions which may be difficult to comprehend and debug.

19.9.5 ▪ The function `free_nodes_list()`

The function `free_nodes_list()` that appears on line 13 of Listing 19.6 on page 232 frees all memory resources associated with a linked list of nodes. Merely applying Chapter 16's `ll_free()` won't suffice since that frees only the memory associated with the cons cells, but the `data` members of the cons cells point to node structures which also take up memory. Therefore, `free_nodes_list()` should traverse the linked list and free those node structures first, and only then apply `ll_free()` to free the cons cells. I leave it to you to implement `free_nodes_list()`.

Normally `free_nodes_list()` is called when the program has completed its work with the analysis of the truss—probably just before it exits—but there are other occasions for it as well. For instance, it is called on line 21 of Listing 19.8 (page 234) to free a partially built list when errors are encountered during the reading of the *TDF*.

19.9.6 ▪ Reading a *TDF*'s *links* section

The functions on lines 20–25 of Listing 19.6 (page 232) are devoted to reading a *TDF*'s *links* section much in the same way that the functions on lines 13–17 were devoted to reading the *nodes* section. The chain of the function calls is similar:

```
read_truss() ⇆ get_links() ⇆ process_link_line() ⇆ make_link()
```

The differences between the link and node reading functions are slight; therefore there is no point in repeating what has been said in subsections 19.9.2 through 19.9.5. I will limit my comments to pointing out a few differences.

The functions `free_links_list()` and `get_links()` basically are copies of the functions `free_nodes_list()` and `get_nodes()`, where all occurrences of "node" change to "link". Note, however, unlike `get_nodes()`, `get_links()` receives a pointer to the linked list of nodes. It has no direct use for that linked list itself; it merely passes it to `process_link_line()`, which in turn passes it to `make_link()`, which needs it for assigning node pointers to the link's endpoints. Once `make_link()` determines those endpoints, it extracts the node coordinates (X_1,Y_1) and (X_2,Y_2) of the

Listing 19.11: A fragment of the function `make_link()` in the file *truss-io.c*.

```
struct link *link;
struct node *np1;
conscell *p;
for (p = nodes_list; p ≠ NULL; p = p→next) {  // look for node number n1
    struct node *node = p→data;
    if (node→n == n1) {
        np1 = node;
        break;
    }
}
if (p == NULL)          // node number n1 not found
    return NULL;
// repeat the same for n2. Then:
link = xmalloc(sizeof *link);
link→np1 = np1;
... etc ...
```

endpoints and calculates the link's natural (undeformed) length, which it stores in the link structure's `L` member; see line 18 in Listing 19.4 (page 230). You will find the preprocessor macro `SQ()` defined on line 10 of Listing 19.6 (page 232) useful for calculating the squares of numbers.

The function `process_link_line()` is similar to `process_node_line()` but simpler since there is only one kind of link description line, "`link n: n1 n2 A E`", while there were four variants of a node description line. It reads the five numbers from the string and passes them to `make_link()`, which sets up an appropriate link structure. It has no direct use for the pointer to the linked list of nodes that it receives as an argument; it merely passes it to `make_link()`, which does. Add an implementation of `process_link_line()` to your *truss-io.c*.

The function `make_link()` is a little more complex than the analogous function `make_node()`. It receives the *node numbers* of a link's end nodes, but what it needs are the *pointers to the corresponding node structures* because that is what goes into a **`struct`** `link`; see Listing 19.4 on page 230. The dilemma is not difficult to resolve. It has received a pointer to the linked list of nodes. It traverses the list, looking for a node with the desired node number. When it finds it, it inserts the node's pointer in the link structure. The code fragment in Listing 19.11 illustrates that.

Flesh out this sketch, and add implementations of `make_link()`, as well as the functions `free_links_list()` and `get_links()`, to your *truss-io.c*.

19.9.7 ▪ Reading a *TDF*'s *loads* section

A *TDF*'s `loads` section is read with the help of the functions `process_load_line()` and `get_loads()` that appear on lines 29 and 30 of Listing 19.6 (page 232). The function `get_loads()` essentially is a copy of `get_links()`, with two differences. First, all occurrences of "link" change to "load". Second, its job is to read the load vector components from the *TDF* and pass them to `process_load_line()` for processing. Unlike `get_links()`, it does not create a new linked list; therefore its returns *true*/*false* (that is, 1 or 0) to indicate success or failure, respectively.

The function `process_load_line()` receives a node number and the load vector components that are applied to it. It searches the linked list of nodes for a node with a

matching node number and deposits the components of the load vector in the node's `fx` and `fy` members, and then it returns 1 (*true*). If a node with the given node number does not exist (this indicates an error in the *TDF*), it returns 0 (*false*). The search for the node is performed exactly like that in `make_link()`.

Add your implementation of `process_load_line()` and `get_loads()` to your *truss-io.c*.

This concludes the description of the details of `read_truss()` and its ancillary functions.

19.9.8 ▪ The function `write_truss()`

The function `write_truss()` that appears on line 35 of Listing 19.6 (page 232) is called after the truss solver (described later) has done its job of determining the truss's deformation. It writes to the solution to the `stdout` in a form conforming to the format of a *TDF* , followed by extra data that give stresses in the links and reaction forces at the supports. Listing 19.2 on page 228 (continued into Listing 19.3 on the next page) shows the output corresponding to the *TDF Cantilever.tdf* of Section 19.6.

The first part of the output, that shown in Listing 19.2, consists of the usual three sections of a *TDF* . In fact, the contents of the *links* and *loads* sections are *identical*—ignoring the cosmetic formatting differences—to those of the input file because link connectivity and applied loads are no different before and after. The contents of the *nodes* section, however, are very different; the coordinates here are those of the truss's *deformed* state. I used those coordinates to draw (manually) the deformed shape in Figure 19.6 on page 222. Those same coordinates are used by the `truss_to_eps()` function of Section 19.10 to automatically generate the drawing shown in Figure 19.10 on page 249.

Remark 19.5. As is noted in the caption of Listing 19.2, the placements of the asterisks against the nodes 0 and 4 are quite different from those seen in the (manually typed) *Cantilever.tdf*, where the asterisks are attached to the respective x and y coordinate values. It is possible to achieve a similar effect on the output; however, the extra work is not worth the effort, so I have left it as it is. Despite that annoying difference, the result still conforms to the *TDF* specifications since `sscanf()` ignores extra whitespace.

The continuation of the output in Listing 19.3 shows additional data regarding the truss's state. The section between "begin reactions" and "end reactions" contains lines of the form `reaction n: rx ry`, where `n` is a node number, and `rx` and `ry` are the components of the support reaction at that node. A support reaction is the force exerted by the support on the truss. Only the support nodes are listed. The calculation of the support reactions is the subject of subsection 19.12.4.

The section between "begin stresses" and "end stresses" contains lines of the form `stress n: s`, where `n` is a link number and `s` is the stress in that link. This is crucial information for truss design—excessive stress will make a link buckle or snap. We will see how to compute the stresses in subsection 19.12.3.

19.9.9 ▪ The function `free_truss()`

The function `free_truss()` that appears on line 36 of Listing 19.6 (page 232) is responsible for freeing all the allocated memory associated with the truss. That memory consists of those of the linked list of the nodes, the linked list of the links, and the truss structure itself. We already have functions to free up the linked lists; therefore the implementation of `free_truss()` is quite trivial:

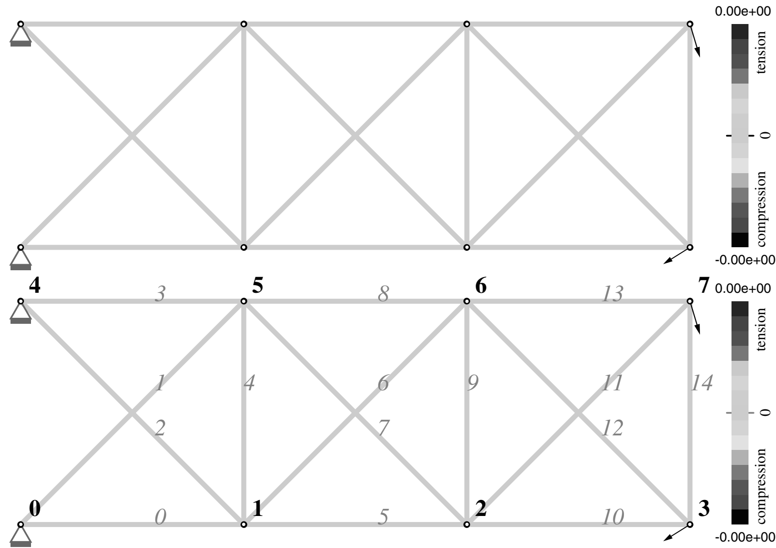


Figure 19.8: Two images of the truss *Cantilever.tdf* (see Section 19.6) in its reference configuration produced by `truss_to_eps()`. The labels in the bottom figure are produced by giving the `TR_PLOT_WITH_LABELS` option in the function's third argument.

```
1  void free_truss(struct truss *truss)
2  {
3      if (truss ≠ NULL) {
4          free_nodes_list(truss→nodes_list);
5          free_links_list(truss→links_list);
6          free(truss);
7      }
8  }
```

The check against `NULL` makes it safe to call `free_truss()` with a `NULL` argument. This is consistent with the standard library's `free()` function, as well as all other `free_*()` functions in this book, which handle a `NULL` gracefully.

Add this implementation of `free_truss()` to your *truss-io.c*.

19.10 ▪ The files *truss-to-eps.[ch]*

I have written a function named `truss_to_eps()` with the prototype

```
void truss_to_eps(struct truss *truss, char *epsfile, int labels);
```

which receives a pointer to a **`struct`** `truss`, from which it extracts the information necessary for drawing an image of the loaded truss, in the EPS format, into a file whose name is specified in the `epsfile` argument. The image depicts the truss, the supports, and the load vectors. The third argument must be one of

```
TR_PLOT_WITH_NO_LABELS   or   TR_PLOT_WITH_LABELS
```

If the `TR_PLOT_WITH_LABELS` is given, the node and link numbers are drawn on the image as well. Figure 19.8 shows images of the truss *Cantilever.tdf* (see Section 19.6) in its reference configuration, with and without labels, produced by `truss_to_eps()`. The deformed image of that same truss is shown in Figure 19.10 on page 249.

Listing 19.12: The file *interlude.c* is a simple driver for testing the `read_truss()` and `write_truss()` functions.

```
1  #include <stdlib.h>
2  #include "truss-to-eps.h"
3  #include "truss-io.h"
4  #include "truss.h"
5  int main(void)
6  {
7      struct truss *truss;
8      if ((truss = read_truss(stdin)) == NULL)
9          return EXIT_FAILURE;
10     truss_to_eps(truss, "z-with-labels.eps", TR_PLOT_WITH_LABELS);
11     truss_to_eps(truss, "z-without-labels.eps", TR_PLOT_WITH_NO_LABELS);
12     write_truss(truss);
13     free_truss(truss);
14     return EXIT_SUCCESS;
15 }
```

Links are drawn in colors that represent the magnitudes of stress in them. Compression is shown in dark red, tension in dark blue, and neutral (zero or almost zero stress) in light green. A color scale accompanying the drawing shows the correspondence between the stress and the varying hues.

The function `truss_to_eps()` consults the values of the (lowercase) `(x,y)` node coordinates[77] to draw the truss. It follows that the resulting image corresponds to the truss's *undeformed shape* if `truss_to_eps()` is called before solving the truss and the *deformed shape* if called after.

Implementing the function `truss_to_eps()` is not a part of this project because that requires some familiarity with the *PostScript* language, which I don't want to make a prerequisite for this book. You should download my implementation of the files *world-to-eps.c* and *world-to-eps.h* from this book's website, drop them in this project's directory, and compile them along with the rest of your files.

19.11 ▪ Interlude (and a mini-project)

At this point we have functions to read and write a *TDF*. It's a good idea to stop for a moment and test those before we move on.

The file *interlude.c* shown in Listing 19.12 is a simple and self-explanatory driver for testing your `read_truss()` and `write_truss()` functions. Compile and link with everything else that it depends on to produce an executable named *interlude*, and then run it:

```
$ ./interlude <Cantilever.tdf >Cantilever.out
```

where *Cantilever.tdf* is as in Section 19.6. If your program works correctly, its output should agree with the *Cantilever.tdf* modulo cosmetic differences. Additionally, it will produce two EPS files named *z-with-labels.eps* and *z-without-labels.eps* which you may view with any *PostScript* viewer. Figure 19.8 shows the results.

For further testing, copy *Cantilever.tdf* to a temporary file, say *tmp.tdf*, introduce deliberate errors of various sorts in it, and verify that your program catches them.

[77] See subsection 19.12.6 for an explanation of the distinction between the `(x,y)` and `(X,Y)` coordinates.

Listing 19.13: An outline of the file *truss.c*.

```
 1  #include <stdio.h>
 2  #include <math.h>
 3  #include "linked-list.h"
 4  #include "array.h"
 5  #include "nelder-mead.h"
 6  #include "truss.h"
 7  #define SQ(x) ((x)*(x))
 8   ▶static double stress_function(double lambda) ...
 9   ▶static double stored_energy_function(double lambda) ...
10   ▶static double link_stretch(struct link *link) ...
11   ▶static void evaluate_stresses(struct truss *truss) ...
12   ▶static void evaluate_reactions(struct truss *truss) ...
13   ▶static double total_energy(double *x, int n, void *params) ...
14   ▶int solve_truss(struct truss *truss,
15               double h, double tol, int maxevals) ...
```

19.12 ▪ The file *truss.c*

The file *truss.c* defines the function `solve_truss()`, which provides the primary user interface to our truss solving utility. Listing 19.13 shows an outline of *truss.c*. Six ancillary functions (lines 8–13) with internal linkage provide support to `solve_truss()`. I will explains the purposes of each of those functions in the following subsections.

19.12.1 ▪ The `stress_function()` and `stored_energy_function()`

The functions `stress_function()` and `stored_energy_function()` that appear on lines 8 and 9 of Listing 19.13 evaluate the template functions $\mathring{\sigma}(\lambda)$ and $\mathring{W}(\lambda)$ defined in equations (19.10) on page 226. Add your implementations of these to your *truss.c*.

19.12.2 ▪ The function `link_stretch()`

The function `link_stretch()` that appears on line 10 of Listing 19.13 receives a pointer to a link structure and calculates the link's *stretch* λ, which, as defined in subsection 19.2.1, is the ratio of the link's deformed and reference lengths.

Since a link structure contains pointers to the nodes at the link's endpoints, you may follow those pointers to access the coordinates `(x,y)` of those nodes and thus calculate the link's deformed length through the usual formula $\sqrt{(x_2-x_1)^2+(y_2-y_1)^2}$. You will find the preprocessor macro `SQ()` defined on line 7 of Listing 19.13 useful for calculating squares of numbers. You won't need to calculate the link's length in the reference configuration—that length is available in the member `L` of the link structure.

Add your implementation of `link_stretch()` to your *truss.c*.

19.12.3 ▪ The function `evaluate_stresses()`

The function `evaluate_stresses()` that appears on line 11 of Listing 19.13 walks through the truss's linked list of links, calculates the stress in each link, and stores it in the link structure's `stress` member. According to (19.10), the stress is $\sigma = E\mathring{\sigma}(\lambda)$. The values of λ and $\mathring{\sigma}(\lambda)$ are computed by calling the functions `link_stretch()` and

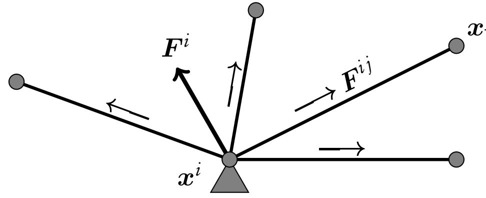


Figure 19.9: Four links connect to, and exert forces on, the support node x^i. Additionally, an external load of F^i is applied directly to the node. The support's reaction is equal and opposite to the vector sum of the forces.

`stress_function()`. The value of Young's modulus E is available as the link structure's `E` member.

Add your implementation of `evaluate_stresses()` to your *truss.c*.

19.12.4 ▪ The function `evaluate_reactions()`

The function `evaluate_reactions()` that appears on line 12 of Listing 19.13 computes the reactions at the support nodes. These are the forces that the supports exert on the truss.

Figure 19.9 shows a support node x^i (marked by a triangle) and the four links[78] that connect to it. In addition to the forces exerted through the links, an external force, F^i, is applied directory to the node.

The link that connects the node x^i to another node, x^j, exerts a force of magnitude $A_{ij}\sigma(\lambda_{ij})$ on the support, where A_{ij} is that link's cross-sectional area and λ_{ij} is its stretch. The force *vector*, which is directed along the link as shown in the figure, is given by

$$F^{ij} = A_{ij}\sigma(\lambda_{ij})\frac{x^j - x^i}{\|x^j - x^i\|}. \tag{19.11}$$

The fraction in the expression above produces the unit vector along the link, directed from x^i to x^j. The support's reaction R^i is equal and opposite the vector sum of all the forces exerted on it:

$$R^i = -\Big(F^i + \sum_{\text{links to } x^i} F^{ij}\Big).$$

You should be able to write your own `evaluate_reactions()` now. Listing 19.14 provides a sketch. Note that in summing the loads on lines 12 and 13 we rely on `node→rx` and `node→ry` being initially zero, which is the case since these were zeroed in the function `make_node()`; see subsection 19.9.4.

Add your implementation of `evaluate_reactions()` to *truss.c*.

19.12.5 ▪ The function `total_energy()`

The function `total_energy()` that appears on line 13 of Listing 19.13 implements the energy of the truss, $\mathcal{E}(x^0, x^1, \dots, x^{n-1})$, defined in (19.9) on page 222. It is shown in Section 19.5 that a minimizer of $\mathcal{E}$ corresponds to an equilibrium position of the truss.

[78] The choice of *four* links is for illustration only. The number of the links is immaterial in the following discussion.

Listing 19.14: A sketch of the function `evaluate_reactions()`.

```
1  step through the linked list of nodes
2      if node is not constrained then
3          continue
4      (otherwise) this is a support node; let x1 = node→x, y1 = node→y
5      step through the linked list of links
6          if link→np1 equals this node, then
7              let (x2,y2) be the coordinates of the other end (i.e., np2)
8          else if link→np2 equals this node, then
9              let (x2,y2) be the coordinates of the other end (i.e., np1)
10         else // link does not connect to this node
11             continue
12         node→rx += ...   // apply (19.11)
13         node→ry += ...   // apply (19.11)
14
15
16     // Done with summing link forces. Add the external load, and then reverse sign.
17     node→rx += node→fx;
18     node→ry += node→fy;
19     node→rx *= -1.0;
20     node→ry *= -1.0;
```

Our program applies the Nelder–Mead simplex method to find that minimizer. The objective function, being the energy in this case, is expected to conform to the prototype of the objective function stipulated in `nelder_mead()`; see Section 18.4 and Listing 18.1. Examining the prototype of `total_energy()` on line 13 of Listing 19.13 we see that it is designed to have exactly the required form.

There is, however, a subtle issue here. The Nelder–Mead algorithm is designed to minimize functions of the type $f : \boldsymbol{R}^n \to \boldsymbol{R}$. Is the energy function $\mathscr{E}(\boldsymbol{x}^0, \boldsymbol{x}^1, \dots, \boldsymbol{x}^{n-1})$ of that type? No! True, it is a function of n variables, but those are not *scalar* variables since every point consists of a pair of coordinates $\boldsymbol{x}^k = (x_k, y_k)$. To appease `nelder_mead()`, we present $\mathscr{E}$ to it not as a function of the n variables $\boldsymbol{x}^0, \boldsymbol{x}^1, \dots, \boldsymbol{x}^{n-1}$ but as a function of the $2n$ "flattened" variables $x_0, y_0, x_1, y_1, \dots, x_{n-1}, y_{n-1}$. Thus, the argument `x` of the function `total_energy()` on line 13 receives the flattened vector $\langle x_0, y_0, x_1, y_1, \dots, x_{n-1}, y_{n-1} \rangle$. It is a part of `total_energy()`'s job to extract the (x_k, y_k) pairs from that representation in order to evaluate the formula (19.9).

With that said, the implementation of `total_energy()`, shown in Listing 19.15, requires little explanation. Note the reconstruction of the node coordinates from the flattened vector on lines 7–11. Be sure to understand what it's doing, and then add it to your *truss.c*.

19.12.6 ▪ The function `solve_truss()`

The function `solve_truss()` that appears on line 14 of Listing 19.13 (page 242) is the only function with external linkage in the file *truss.c*. Its purpose is to compute a local minimum of the energy of the truss that it receives as an argument. It calls upon `nelder_mead()` to perform the minimization. The values of the arguments `h`, `tol`, and `maxevals` which it passes to `nelder_mead()` determine the precision and stopping criteria. You will have to read Chapter 18 for the details of the meanings of those parameters.

Listing 19.15: The function `total_energy()` in the file *truss.c* evaluates the truss's total energy, $\mathscr{E}(\boldsymbol{x}^0, \boldsymbol{x}^1, \dots, \boldsymbol{x}^{n-1})$. The argument `x` is the flattened vector $\langle x_0, y_0, x_1, y_1, \dots, x_{n-1}, y_{n-1}\rangle$.

```
1  static double total_energy(double *x, int n, void *params)
2  {
3      struct truss *truss = params;
4      conscell *p;
5      double e = 0.0;
6      int j = 0;
7      for (p = truss->nodes_list; p != NULL; p = p->next) {
8          struct node *node = p->data;
9          node->x = x[j++];
10         node->y = x[j++];
11     }
12     for (p = truss->links_list; p != NULL; p = p->next) {
13         struct link *link = p->data;
14         e += link->E * link->A * link->L
15             * stored_energy_function(link_stretch(link));
16     }
17     for (p = truss->nodes_list; p != NULL; p = p->next) {
18         struct node *node = p->data;
19         e -= (node->x - node->X) * node->fx
20             + (node->y - node->Y) * node->fy;
21     }
22     return e;
23 }
```

The position of the initial simplex is a crucial factor in determining the outcome of the Nelder–Mead iterations. If the objective function has multiple local minima, as does a truss's energy function in general, the algorithm is likely to converge to the local minimum that is closest to the starting point.

Our solver's default choice for the initial simplex corresponds to the truss's reference configuration, but the user may override that default if so desired. To understand this fully, we need to take a closer look at the role of a node's coordinates.

Let us recall that the **`struct`** `node` structure maintains two set of coordinates, `(X,Y)` and `(x,y)`, for each node. The `(X,Y)` coordinates hold the node's position in the truss's reference configuration. These are assigned once and don't change for the duration of the program. The `(x,y)` coordinates are initialized to `(X,Y)`; see subsection 19.9.4. After the truss is solved, they take on the values of the node's deformed position.

The program builds the Nelder–Mead initial simplex based on the values of the (lowercase) `(x,y)` coordinates. Since these are initialized to coincide with the (uppercase) `(X,Y)`, the default initial simplex corresponds to the truss's reference configuration. The user has the opportunity to alter the values of `(x,y)` at will between calls to `read_truss()` and `solve_truss()`, and thereby to set the initial simplex near a desired target position. See Part 19.4 in this chapter's *Projects* section for an example.

Listing 19.16 shows my implementation of `solve_truss()`. Here I will comment on some of its more interesting features.

On line 6 the dimension `N` of the space is set to *twice* the number of nodes. This is because the objective function, that is, the energy, is a function of that many variables; see the discussion about the "flattened" variables in the previous subsection.

Listing 19.16: The function `solve_truss()` in the file *truss.c*.

```
int solve_truss(struct truss *truss, double h, double tol, int maxevals)
{
    double **s;
    double *x;
    conscell *p;
    int N = 2*truss->nnodes;        // dimension of the energy space
    struct nelder_mead NM;
    int i, j, evalcount;
    make_vector(x, N);
    for (j = 0, p = truss->nodes_list; p != NULL; p = p->next) {
        struct node *node = p->data;
        x[j++] = node->x;
        x[j++] = node->y;
    }
    make_matrix(s, N + 1, N);      // the simplex matrix
    for (i = 0; i < N + 1 ; i++)
        for (j = 0; j < N; j++)
            s[i][j] = x[j];
    for (j = 0; j < N; j++)
        s[j+1][j] += h;
    // project the simplex over the constraint space
    for (j = 0, p = truss->nodes_list; p != NULL; p = p->next) {
        struct node *node = p->data;
        if (node->xfixed)
            for (i = 0; i < N + 1 ; i++)
                s[i][j] = node->X;
        j++;
        if (node->yfixed)
            for (i = 0; i < N + 1 ; i++)
                s[i][j] = node->Y;
        j++;
    }
    NM.f = total_energy;
    NM.n = N;
    NM.s = s;
    NM.x = x;
    NM.h = h;
    NM.tol = tol;
    NM.maxevals = maxevals;
    NM.params = truss;
    evalcount = nelder_mead(&NM);
    free_vector(x);
    free_matrix(s);
    if (evalcount > maxevals) {
        fprintf(stderr, "No convergence after %d "
                "function evaluation\n", evalcount);
        return 0;
    } else {
        fprintf(stderr, "Nelder-Mead converged after %d "
                "function evaluations\n", evalcount);
        evaluate_stresses(truss);
        evaluate_reactions(truss);
        return 1;
    }
}
```

Line 7 defines NM as an instance of a **struct** nelder_mead. This will bundle all the information required for the minimization and pass that to nelder_mead() for processing.

The action begins on line 9, where a vector x of length N is created to serve as the calculation's independent variable. Recall that N is twice the number of nodes.

The **for**-loop on lines 10–14 flattens the energy function's arguments—it traverses the linked list of the truss's nodes, reads the coordinates node→x and node→y of each node, and enters them as consecutive entries in the vector x.

Lines 15–20 create the (N+1) × N simplex matrix and populate its rows with copies of the vector x first, and then they adjust them according to (18.5) on page 200 to form a simplex of size h.

The coordinates of some of the truss's support nodes may be completely fixed (in an anchor node) or partially fixed (in a rolling node). Therefore, although the energy is a function of N variables $\langle x_0, y_0, x_1, y_1, \dots, x_{n-1}, y_{n-1} \rangle$, not all of those variables are free to change. Thus, the minimization of the energy takes place not on an N-dimensional space, but on an affine subspace defined by constraints of the form $x_k = \mathring{x}_k$ or $y_k = \mathring{x}_y$, for certain indices k, where $\mathring{x}_k$ and $\mathring{x}_y$ are fixed.

Minimizing functions subject to affine constraints is explained in Section 18.8. The outcome of that section's discussion is that if we project an arbitrary initial simplex onto the constraint subspace, the Nelder–Mead algorithm will ensure that its subsequence iterates remain within that subspace. Furthermore, Remark 18.9 on page 212 points out that projecting onto constraint spaces of the form $x_k = \mathring{x}_k$ is particularly simple; it's a matter of setting every entry in the corresponding column of the simplex matrix equal to $\mathring{x}_k$. This is precisely what is done on lines 22–32 of Listing 19.16. Examine that code, and make sure that you understand its details.

Now all is ready for calling the Nelder–Mead minimizer. On lines 33–40 we set up the members of the NM structure; then on line 41 we pass the address of that structure to nelder_mead(). Once nelder_mead() returns, there is no longer a need for the vector x and matrix s, so we free them on lines 42 and 43.

There is a chance, of course, that Nelder–Mead fails to converge. We have designed nelder_mead() so that it halts if the number of function evaluations exceeds the maximum prescribed in maxevals. On line 44 we check the evaluation count against maxevals. If it is more, then we conclude that the algorithm hasn't converged, so we print a message to that effect to stderr and return 0, that is, *false*, to the caller. Otherwise, we print an a informative message regarding success (again, to stderr), then we call the postprocessing functions evaluate_stresses() and evaluate_reactions() to compute the stresses in the links and the reaction forces at the supports, and finally we return 1, that is, *true*, to the caller.

19.13 ▪ The file *truss-demo.c*

The file *truss-demo.c* shown in Listing 19.17 provides a driver for demonstrating, and experimenting with, our truss solver. It is quite short, and much of it is self-explanatory; nevertheless, I will make a few comments on it.

Line 11. The function read_truss() is invoked to read a *TDF* from the stdin. If the call fails, as it will if the *TDF* contains incomplete or inconsistent data, it returns NULL, in which case the program exits with an EXIT_FAILURE status. We need not print a message here since read_truss() prints its own diagnostics. If the

Listing 19.17: The file *truss-demo.c* is a demo driver for the truss solver utility.

```
#include <stdlib.h>
#include "truss-to-eps.h"
#include "truss-io.h"
#include "truss.h"
int main(void)
{
    double tol = 1.0e-3;
    double h   = 0.1;
    int maxevals = 50000;
    struct truss *truss;
    if ((truss = read_truss(stdin)) == NULL)
        return EXIT_FAILURE;
    // write undeformed configuration image to file z1.eps
    truss_to_eps(truss, "z1.eps", TR_PLOT_WITH_LABELS);
    if (!solve_truss(truss, h, tol, maxevals)) {
        free_truss(truss);
        return EXIT_FAILURE;
    }
    write_truss(truss);
    // write deformed configuration image to file z2.eps
    truss_to_eps(truss, "z2.eps", TR_PLOT_WITH_NO_LABELS);
    free_truss(truss);
    return EXIT_SUCCESS;
}
```

call succeeds, then `read_truss()` returns a pointer to a **struct** `truss` where it has stored the data fetched from the *TDF* .

Line 14. The pointer to the truss structure is passed to the function `truss_to_eps()` to write an image of the truss's reference (undeformed) configuration into an EPS file named *z1.eps*. (The name of the file is arbitrary; name it anything you wish.) The argument `TR_PLOT_WITH_LABELS` requests that the image include node and link labels. This results in the second of the two images in Figure 19.8 on page 240.

Line 15. The function `solve_truss()` is called to solve the truss, that is, to find its deformed configuration. The arguments `h`, `tol`, and `maxevals` control the behavior of the Nelder–Mead simplex which is used to minimize the truss's total energy. If `solve_truss()` fails, the program frees the previously allocated memory and then exits with the `EXIT_FAILURE` status.

Line 19. The function `write_truss()` is called to print the details of the solved truss to the `stdout`.

Line 21. The function `truss_to_eps()` is called for a second time to write the image of the now deformed truss to the EPS file *z2.eps*. The argument `TR_PLOT_WITH_NO_LABELS` requests that the image *not* include node and link labels. Figure 19.10 shows the result.

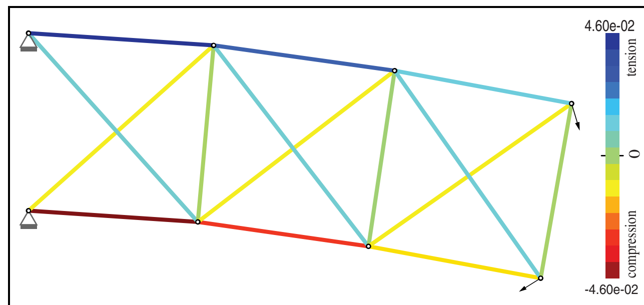


Figure 19.10: The truss *Cantilever.tdf* deformed under the applied loads, as drawn by `truss_to_eps()`.

Finally the program calls `free_truss()` to free all allocated memory and then exits with an `EXIT_SUCCESS` status.

19.14 ▪ *Project Truss*

Part 19.1. Complete and test the mini-project of Section 19.11.

Part 19.2. Complete the rest of the program described in this chapter, and run tests with the supplied *Cantilever.tdf* and *Pratt.tdf*. Vary the applied loads to see whether the truss responds as expected. Note that the driver *truss-demo.c* in Listing 19.17 writes the truss's reference configuration to a file named *z1.eps* and the deformed configurations to a file named *z2.eps*. You may wish to change these to more meaningful names.

Part 19.3. Write a *TDF* that describes the two-link truss shown on page 207. Set the Young's modulus, E, and cross-sectional area, A, to 1 for both links, and apply a load of $\langle 0, -0.3 \rangle$ to the node C. Under these conditions, there is a single stable equilibrium corresponding to a global minimum D_1, as shown in the top-left drawing of Figure 18.2. Apply the driver *truss-demo.c* to find the coordinates of D_1 and the shape of the deformed truss.

Answer: $D_1 : (0.8319642234, -1.250527826)$.

Part 19.4. Repeat the previous part, but change the load to $\langle 0, -0.1 \rangle$. Under these conditions, there are two stable equilibria corresponding to the local minima D_1 and D_2, as shown in the top-right drawing of Figure 18.2. These are:

$$D_1 : (0.9208762302, -1.092612810), \qquad D_2 : (1.155467267, 0.8776603420).$$

Verify that applying the driver *truss-demo.c* finds the minimum at D_2. Refer to subsection 19.12.6 for an explanation of why the default search converges to a configuration closest to the truss's undeformed configuration.

Part 19.5. [Optional] We wish to repeat the previous part but pick up the minimum at D_1 instead. To that end, you will have to start the Nelder–Mead simplex near D_1. It's best to copy the driver *truss-demo.c* to a new file, let's call it *truss-demo-two-links.c*, and modify it as follows: After calling `read_truss()` but before `solve_truss()`, change the

(lowercase) `(x,y)` coordinates of the node at vertex C (see the drawing in Figure 18.2) to something like (0.5, −1.0).

Part 19.6. [Optional] Our program deals with planar (two-dimensional) trusses. Many trusses in real applications are three-dimensional, e.g., the boom of a construction crane or a high-voltage power transmission tower. Write a version of the truss solver for three-dimensional trusses. For that, copy *truss-io.[ch]* and *truss.[ch]* to *truss3D-io.[ch]* and *truss3D.[ch]*, and then edit the new files to make them handle the extra dimension. You will find out that the required changes are few and quite trivial. Unfortunately you will lose the use of `truss_to_eps()` since that's strictly for plotting two-dimensional trusses. *Geomview* would be a good substitute for the three-dimensional case. You will have to study its documentation to learn about the format of its graphics files.

Chapter 20

Finite difference schemes for the heat equation in one dimension

Prerequisites: *Chapters 7, 8*

20.1 ▪ The basic idea of finite differences

In this chapter we apply a variety of finite difference techniques to approximate the solutions of initial/boundary value problems associated with the *heat equation*

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}. \tag{20.1}$$

The unknown $u = u(x,t)$ is a function of space x and time t.

The partial differential equation (20.1) arises in a variety of contexts in mathematical physics, probability theory, digital image processing, chemistry, and financial mathematics. Perhaps the most accessible instance is as a model of the temperature in a one-dimensional heat-conducting rod which is thermally insulated all around except for its ends, where it interacts with the outside world. If we hold a flame to one end, heat will propagate through the rod and affect the temperature everywhere. The function $u(x,t)$ is the temperature at the point x at time t. The partial differential equation (20.1) accounts for the conservation of thermal energy within the rod.

We obtain a well-posed heat conduction problem if we specify the rod's temperature at time zero, that is, $u(x,0)$, and prescribe the temperatures at its ends at all times, that is, in $u(a,t)$ and $u(b,t)$. Here I am assuming that the rod coincides with the interval (a,b) on the x axis. This information, along with (20.1), should suffice to determine the temperature $u(x,t)$ at all points $x \in (a,b)$ and all times $t > 0$. We state this formally as the following initial/boundary value problem:

Find $u = u(x,t)$ so that

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}, \qquad x \in (a,b),\ t > 0, \tag{20.2a}$$

$$u(x,0) = u_0(x), \qquad x \in (a,b), \tag{20.2b}$$

$$u(a,t) = u_L(t), \quad u(b,t) = u_R(t), \qquad t > 0. \tag{20.2c}$$

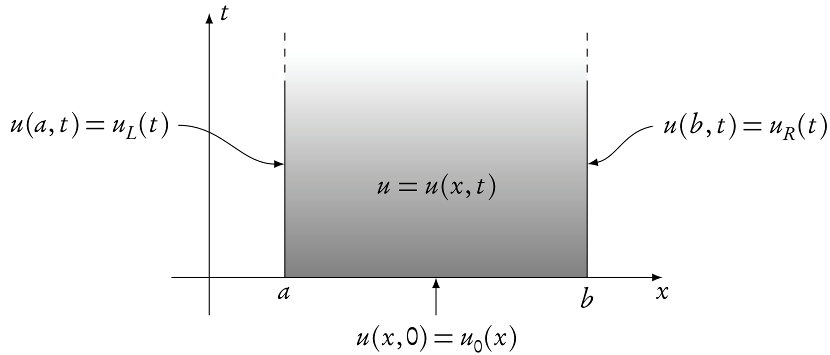


Figure 20.1: The initial condition $u_0(x)$ and boundary conditions $u_L(t)$ and $u_R(t)$ determine the solution of the heat equation in the shaded semi-infinite strip.

The *initial condition* $u_0(x)$ and the left and right *boundary conditions* $u_L(t)$ and $u_R(t)$ are prescribed. They serve to define a unique[79] solution $u(x,t)$ in the semi-infinite strip $a \le x \le b$ and $t > 0$ in the x-t plane. See Figure 20.1.

In the finite differences worldview, space and time are discrete. The interval $a \le x \le b$ is replaced by a collection of $n+2$ points $x_0 < x_1 < \cdots < x_{n+1}$, where $x_0 = a$ and $x_{n+1} = b$. We call x_0 and x_{n+1} the *boundary points* and the remaining n, that is, $x_1, x_2, \ldots, x_n$, the *internal points*. We assume, for simplicity's sake, that the $n+2$ points are evenly spaced, and thus they partition the interval $[a,b]$ into $n+1$ subintervals of length $\Delta x = (b-a)/(n+1)$ each. Consequently, $x_j = a + j\Delta x$, $j = 0, 1, \ldots, n+1$.

Similarly, the time is discretized into "time-slices" $t_0 < t_1 < t_2 \cdots$, where $t_0 = 0$. We assume that the time-slices are evenly spaced at a prescribed Δt intervals, and therefore $t_k = k\Delta t$, $k = 0, 1, 2, \ldots$.

The discretization of the space and time replaces the shaded strip of Figure 20.1 by a grid of points (x_j, t_k), as depicted in Figure 20.2. The task of finding the function $u(x,t)$ is replaced by the task of computing its values $u(x_j, t_k)$ at the grid points. For convenience we introduce the notation u_j^k for $u(x_j, t_k)$ since it is more compact and easier to parse. I trust that it's clear that k is a superscript here, not an exponent!

The derivative $\partial u/\partial t$ at (x_j, t_k) may be approximated by either of the following two ways:

$$\left.\frac{\partial u}{\partial t}\right|_{(x_j,t_k)} \approx \frac{u_j^{k+1} - u_j^k}{\Delta t} \qquad \text{or} \qquad \left.\frac{\partial u}{\partial t}\right|_{(x_j,t_k)} \approx \frac{u_j^k - u_j^{k-1}}{\Delta t}. \tag{20.3}$$

The first variant is called a *forward difference approximation* since it looks up the value of u at a future time. The second variant is called a *backward difference approximation* since it looks up the value of u at a previous time. Both have their uses, as we shall see.

To approximate the second derivative $\partial^2 u/\partial x^2$ at (x_j, t_k), let us look at the Taylor expansion of $u(x, t_k)$ about $x = x_j$:

$$u(x,t_k) = u(x_j,t_k) + \left.\frac{\partial u}{\partial x}\right|_{(x_j,t_k)} (x - x_j) + \left.\frac{\partial^2 u}{\partial x^2}\right|_{(x_j,t_k)} (x - x_j)^2 + \cdots.$$

[79] I am hiding some technical details here. The existence and uniqueness of a solution u depend on the regularity and integrability of the functions u_0, u_L, u_R; see e.g., Friedman [21]. Those details, however, hardly matter for the purposes of this chapter.

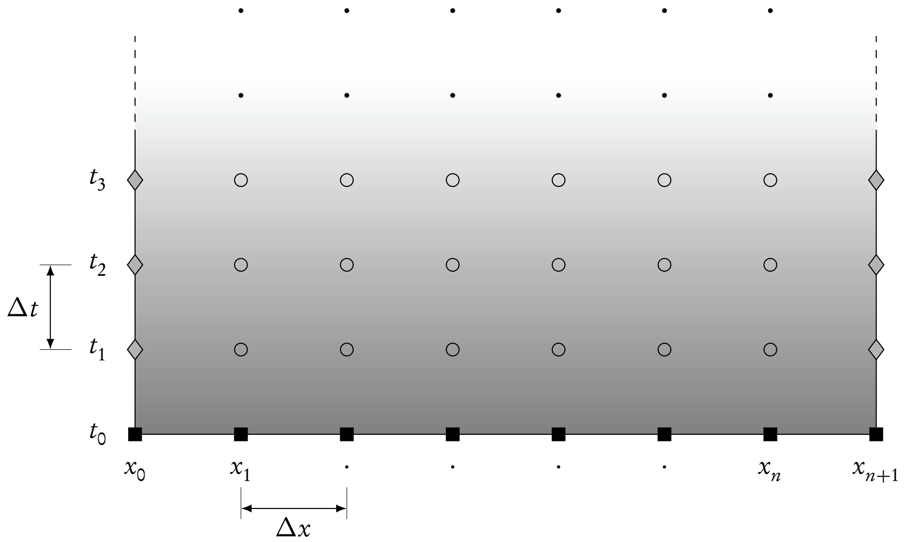


Figure 20.2: The finite difference grid consists of $n+2$ points $x_0, x_1, \ldots, x_{n+1}$ in the x direction and a sequence of time-slices $t_0, t_1, t_2, \ldots$ in the t direction. The initial condition determines the solution at the squares ■. The boundary conditions determine the solution at the diamonds ◊. The finite difference algorithm determines the solution at the rest of the grid points marked with the hollow circles ○.

Evaluating this at $x = x_{j-1}$ and $x = x_{j+1}$ we get

$$u(x_{j-1}, t_k) = u(x_j, t_k) + \frac{\partial u}{\partial x}\bigg|_{(x_j,t_k)} (x_{j-1} - x_j) + \frac{\partial^2 u}{\partial x^2}\bigg|_{(x_j,t_k)} (x_{j-1} - x_j)^2 + \cdots,$$

$$u(x_{j+1}, t_k) = u(x_j, t_k) + \frac{\partial u}{\partial x}\bigg|_{(x_j,t_k)} (x_{j+1} - x_j) + \frac{\partial^2 u}{\partial x^2}\bigg|_{(x_j,t_k)} (x_{j+1} + x_j)^2 + \cdots.$$

We then substitute $x_{j+1} + x_j = \Delta x$ and $x_{j-1} + x_j = -\Delta x$, add up the resulting equations, switch to the compact notation u_j^k introduced above, and arrive at

$$u_{j-1}^k + u_{j+1}^k = 2u_j^k + \frac{\partial^2 u}{\partial x^2}\bigg|_{(x_j,t_k)} (\Delta x)^2 + \cdots,$$

whence

$$\frac{\partial^2 u}{\partial x^2}\bigg|_{(x_j,t_k)} \approx \frac{u_{j-1}^k - 2u_j^k + u_{j+1}^k}{(\Delta x)^2}. \tag{20.4}$$

The approximations in (20.3) and (20.4) are the main tools in the field of finite differences. They may be applied in a variety of ways to approximate the problem (20.2) with discrete versions. We will study a few possibilities in the following sections. To learn more about the subject, you may start with one of [71, 33, 29].

20.2 ▪ An explicit scheme for the heat equation

In the heat equation (20.2a), replace the right-hand side by the approximation given in (20.4) and the left-hand side by the *forward difference approximation* defined in (20.3).

We obtain

$$\frac{u_j^{k+1} - u_j^k}{\Delta t} = \frac{u_{j-1}^k - 2u_j^k + u_{j+1}^k}{(\Delta x)^2}, \quad j = 1, 2, \ldots, n.$$

We see that the space and time increments, Δx and Δt, enter in the form of the combination $\Delta t/(\Delta x)^2$; therefore it makes sense to introduce the notation

$$r = \frac{\Delta t}{(\Delta x)^2} \tag{20.5}$$

and express the equation in terms of r, as in

$$u_j^{k+1} - u_j^k = r(u_{j-1}^k - 2u_j^k + u_{j+1}^k), \quad j = 1, 2, \ldots, n, \tag{20.6}$$

or the rearranged form

$$u_j^{k+1} = r u_{j-1}^k + (1 - 2r) u_j^k + r u_{j+1}^k, \quad j = 1, 2, \ldots, n. \tag{20.7}$$

The result is very revealing. It says that u_j^{k+1}, that is, the value of u at the time-slice $k+1$, may be computed from the values of u at the time-slice k. Since the values of u_j^k at the time-slice $t = 0$ are known—that's what the initial condition is for—we may apply (20.7) recursively, one time-slice at a time, to march forward through the time-slices and determine u_j^k at all times. If you mark the formula's entries on the finite difference grid, as is done in two instances in Figure 20.3, they arrange themselves in a $\perp$-shaped pattern. That pattern, which is a characteristic of the finite difference scheme (20.7), is called the scheme's *stencil*. If the stencil contacts the left or right boundary, as it has in one of the instances shown, it picks up the user-supplied boundary condition there. That's where equations (20.2c) come in.

The recursion formula (20.7) is called an *explicit scheme* since it provides the value of u_j^{k+1} explicitly, with no fuss, in terms of given or previously computed data. In that sense it is quite trivial to implement it in a program—just put the formula in a **`for`**-loop and compute away. We will do exactly that in our implementation. For the conceptual understanding and analysis, however, the matrix form of that formula provides a deeper insight. To obtain the matrix form, it helps to write out several instances of the formula explicitly,

$$\begin{aligned}
j = 1: &\qquad u_1^{k+1} = r u_0^k + (1 - 2r) u_1^k + r u_2^k, \\
j = 2: &\qquad u_2^{k+1} = r u_1^k + (1 - 2r) u_2^k + r u_3^k, \\
\ldots &\qquad \ldots \\
j = n: &\qquad u_n^{k+1} = r u_{n-1}^k + (1 - 2r) u_n^k + r u_{n+1}^k,
\end{aligned}$$

and then, after letting $s = 1 - 2r$, pack the equations into a matrix-vector form:

$$\begin{pmatrix} u_1^{k+1} \\ u_2^{k+1} \\ \vdots \\ u_{n-1}^{k+1} \\ u_n^{k+1} \end{pmatrix} = \begin{pmatrix} s & r & & & \\ r & s & r & & \\ & \ddots & \ddots & \ddots & \\ & & r & s & r \\ & & & r & s \end{pmatrix} \begin{pmatrix} u_1^k \\ u_2^k \\ \vdots \\ u_{n-1}^k \\ u_n^k \end{pmatrix} + \begin{pmatrix} r u_0^k \\ 0 \\ \vdots \\ 0 \\ r u_{n+1}^k \end{pmatrix}. \tag{20.8}$$

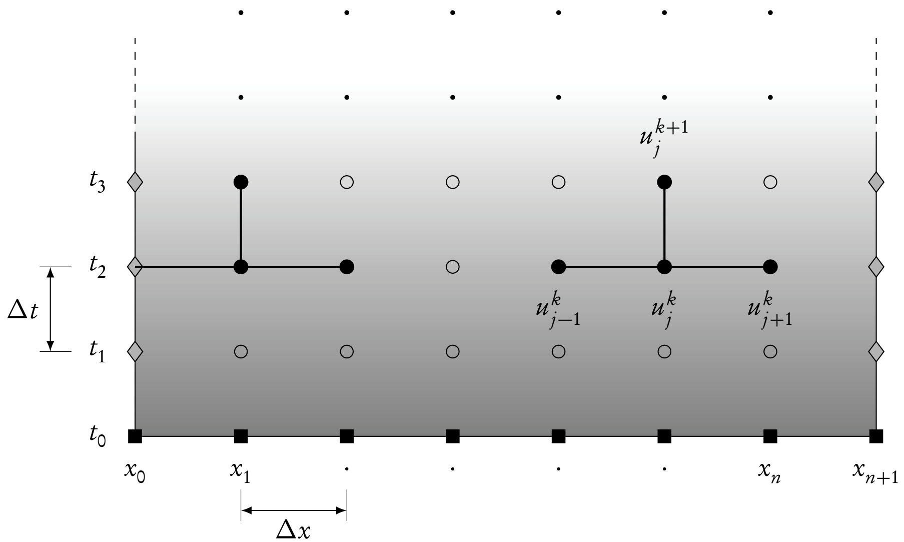


Figure 20.3: The explicit finite difference scheme acts on a ⊥-shaped stencil. It determines the values of u_j^{k+1} at the time-slice $k+1$ in terms of the three values of u from the previous time-slice. When the stencil hits the left or right boundary, its picks up the prescribed boundary value from there.

The matrix is tridiagonal, having $s = 1-2r$ on its main diagonal and r on its first upper and lower subdiagonals. All other entries are zeros. We note that the matrix acts on the grid's internal nodes at the time-slice k. The additive vector in the equation's extreme right imports the prescribed data from the boundary nodes x_0 and x_{n+1}. In effect, the equation defines a transition operator that maps the solution from the time-slice k to the time-slice $k+1$.

The recursion scheme (20.7), or its matrix equivalent (20.8), provides an extremely quick and simple method for solving the heat equation and its relatives (general parabolic equations). Of course it is natural to ask the following: Does it produce a good approximation? Does the approximation improve as Δx and Δt go to zero? It turns out that the answers to both questions are a qualified "yes". The catch is that you cannot take Δx and Δt entirely independently of each other. The method is guaranteed to work only if $r \le 1/2$, where r is defined in (20.5).

To get a feel for the source of the trouble, consider the special case where the boundary conditions $u_L(t)$ and $u_R(t)$ in (20.2c) are zero. Then we expect the rod's temperature to go to zero in the long run, regardless of the initial condition, since its ends are being kept at zero temperature. The finite difference scheme should confirm that. But does it?

When the boundary conditions are zero, the iteration scheme in (20.8) takes the form $\boldsymbol{u}^{k+1} = A\boldsymbol{u}^k$, where A is the tridiagonal matrix in that equation, and $\boldsymbol{u}^k$ is the vector that it multiplies. We see that $\boldsymbol{u}^1 = A\boldsymbol{u}^0$, $\boldsymbol{u}^2 = A\boldsymbol{u}^1$, etc., and consequently $\boldsymbol{u}^k = A^k\boldsymbol{u}^0$. It can be shown (see [33], for instance) that the eigenvalues of A are

$$\lambda_j = 1 - 2r\cos\frac{j\pi}{n+1}, \quad j = 1, 2, \ldots, n. \tag{20.9}$$

Therefore, if $r \le 1/2$, then all eigenvalues are strictly less than 1, and consequently $A^k \to 0$ as $k \to \infty$, confirming our expectation. If $r > 1/2$, however, and n is sufficiently large,

then it is possible for some of the eigenvalues to be greater than 1; therefore $A^k \boldsymbol{u}^0$ will grow unbounded as $k \to \infty$, at least for some choices of $\boldsymbol{u}^0$. That is a totally unreasonable way for heat to behave; the iteration is producing junk!

An iteration scheme is said to be *stable* if small perturbations in the problem's data result only in small perturbations in the solution. Otherwise it is said to be *unstable*. The explicit finite difference scheme defined by (20.7)—or the equivalent (20.8)—is *conditionally stable*: it is stable when $r \le 1/2$.

Why should conditional stability concern us? Even with the most generous choice of $r = 1/2$, (20.5) tells us that $\Delta t = \frac{1}{2}(\Delta x)^2$. This implies that if Δx is small, then Δt will be uncomfortably small. For instance, if $\Delta x = 1/10$, then $\Delta t = 1/200$. Thus, to compute the solution up to time $t = 1$, we will have to march through 200 time-slices. If we change Δx to $1/100$ to achieve a higher accuracy, then Δt changes to $1/20,000$, forcing us to plod through 20,000(!) time-slices to traverse the time interval 0 to 1. That's an inordinate amount of work.

The implicit finite difference scheme, introduced in the next section, removes that restriction on r.

20.3 ▪ An implicit scheme for the heat equation

In the heat equation (20.2a), replace the right-hand side by the approximation given in (20.4) and the left-hand side by the *backward difference approximation* defined in (20.3). We obtain

$$\frac{u_j^k - u_j^{k-1}}{\Delta t} = \frac{u_{j-1}^k - 2u_j^k + u_{j+1}^k}{(\Delta x)^2}, \quad j = 1, 2, \dots, n.$$

For notational consistency with the previous section, shift the superscript k up by one, and also set $r = \Delta t/(\Delta x)^2$, as before, to get

$$u_j^{k+1} - u_j^k = r(u_{j-1}^{k+1} - 2u_j^{k+1} + u_{j+1}^{k+1}), \quad j = 1, 2, \dots, n. \tag{20.10}$$

Then rearrange/group terms to arrive at

$$-r u_{j-1}^{k+1} + (1+2r)u_j^{k+1} - r u_{j+1}^{j+1} = u_j^k, \quad j = 1, 2, \dots, n. \tag{20.11}$$

The formula's entries form a ⊤-shaped stencil on the finite difference grid. Two instances of the stencil are shown in Figure 20.4. If the stencil contacts the left or right boundary, as it has in one of the instances shown, it picks up the user-supplied boundary condition there.

On the surface, this looks very similar to the previous section's (20.7). There is, however, something fundamentally different here. Equation (20.7) expresses u at the time-slice $k+1$ explicitly in terms of the values of u at the previous time-slice. Equation (20.11), however, does not do that. It expresses a certain *combination* of the values of u at the time-slice $k+1$ in terms of u at the previous time-slice. It is called an *implicit scheme* for that reason.

To grasp fully what the implicit scheme (20.11) represents, let us write it out in detail:

$$\begin{aligned}
j = 1: &\qquad -r u_0^{k+1} + (1+2r)u_1^{k+1} - r u_2^{k+1} = u_1^k, \\
j = 2: &\qquad -r u_1^{k+1} + (1+2r)u_2^{k+1} - r u_3^{k+1} = u_2^k, \\
\cdots &\qquad \cdots \\
j = n: &\qquad -r u_{n-1}^{k+1} + (1+2r)u_n^{k+1} - r u_{n+1}^{k+1} = u_n^k.
\end{aligned}$$

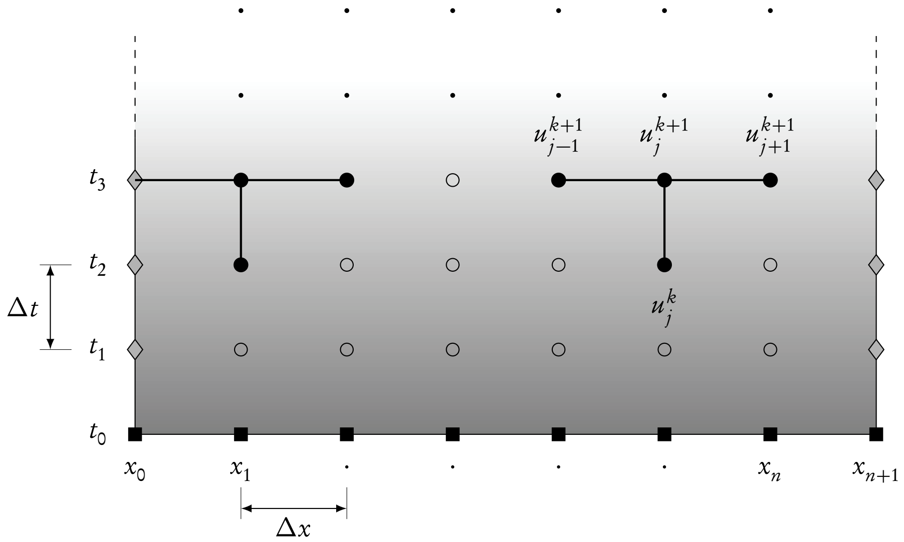


Figure 20.4: The implicit finite difference scheme acts on a ⊤-shaped stencil. It relates a linear combination of three values of u_j^{k+1} at the time-slice $k+1$ to a value of u from the previous time-slice. When the stencil hits the left or right boundary, its picks up the prescribed boundary value from there.

Then, after letting[80] $s = 1+2r$, pack it into a matrix-vector form:

$$\begin{pmatrix} s & -r & & & \\ -r & s & -r & & \\ & \ddots & \ddots & \ddots & \\ & & -r & s & -r \\ & & & -r & s \end{pmatrix} \begin{pmatrix} u_1^{k+1} \\ u_2^{k+1} \\ \vdots \\ u_{n-1}^{k+1} \\ u_n^{k+1} \end{pmatrix} = \begin{pmatrix} u_1^k \\ u_2^k \\ \vdots \\ u_{n-1}^k \\ u_n^k \end{pmatrix} + \begin{pmatrix} r\,u_0^{k+1} \\ 0 \\ \vdots \\ 0 \\ r\,u_{n+1}^{k+1} \end{pmatrix}. \tag{20.12}$$

The matrix is tridiagonal, having $s = 1+2r$ on its main diagonal and $-r$ on its first upper and lower subdiagonals. All other entries are zeros. As a whole, the equation relates the state of the solution at the grid's internal nodes at the time-slice $k+1$ to those at the time-slice k. The additive vector in the equation's extreme right imports the prescribed data from the boundary nodes x_0 and x_{n+1}.

Computing the solution at the time-slice $k+1$ calls for solving the system of linear equations (20.12). The task is quite simple on account of the coefficient matrix being tridiagonal. We will study the solution algorithm in subsection 20.9.3. For now, let us look at the scheme's stability.

As in the previous section, consider the special case where the boundary conditions $u_L(t)$ and $u_R(t)$ in (20.2c) are zero. Then the iteration scheme in (20.12) takes the form $B\boldsymbol{u}^{k+1} = \boldsymbol{u}^k$, where B is the tridiagonal matrix. We see that $B\boldsymbol{u}^1 = \boldsymbol{u}^0$, $B\boldsymbol{u}^2 = \boldsymbol{u}^1$, etc., and consequently $B^k\boldsymbol{u}^k = \boldsymbol{u}^0$, and therefore $\boldsymbol{u}^k = B^{-k}\boldsymbol{u}^0$.

Considering that $s = 1+2r$ in this section while $s = 1-2r$ in the previous section, it should be evident that the matrix B here is related to the previous section's matrix A through the change of variables $r \to -r$. Therefore B's eigenvalues are obtained by

[80] Beware that $s = 1+2r$ is different from the previous definition of s.

changing r to $-r$ in (20.9). Thus, B^{-1}'s eigenvalues are

$$\lambda_j = \frac{1}{1+2r\cos\frac{j\pi}{n+1}}, \quad j=1,2,\ldots,n.$$

We see that these are all strictly less than 1 regardless of the value of r (we are taking it for granted that $r > 0$). It follows that $B^{-k} \to 0$ as $k \to \infty$; therefore the iteration scheme (20.12) is stable for all r. One expresses this by saying that the scheme is *unconditionally stable*.

Unconditional stability is a *good thing* since it removes the worry about the right choice of r. But lest you jump to unwarranted conclusions, let me point out that things are not quite as rosy as you might expect.

Toward the end of the previous section, when discussing the stability of the explicit scheme, I noted that the requirement $r \le 1/2$ is quite inconvenient since $\Delta t = r(\Delta x)^2$ forces exceedingly small time-steps when Δx is somewhat small. The implicit scheme, however, imposes no restriction on r, so one may be tempted to take arbitrarily large time-steps Δt, uncoupled from the size of Δx. This, however, does not work as well as we may wish. It can be shown (see [33], for instance) that the discretization error in the implicit scheme is of the order of magnitude of $\Delta t + (\Delta x)^2$. For best results we want to keep Δt and $(\Delta x)^2$ in more or less comparable sizes; otherwise the larger of the two will dominate the error. This, in effect, limits Δt to something of the order of magnitude of $(\Delta x)^2$ even though there is no restriction on r. The implicit scheme, therefore, has not released us from the bind of small time-steps.

The Crank–Nicolson scheme, introduced in the next section, gets around this dilemma.

20.4 ▪ The Crank–Nicolson scheme for the heat equation

The Crank–Nicolson scheme is obtained by summing the explicit and implicit differencing formulas (20.6) and (20.10),

$$2(u_j^{k+1} - u_j^k) = r\big(u_{j-1}^k - 2u_j^k + u_{j+1}^k + u_{j-1}^{k+1} - 2u_j^{k+1} + u_{j+1}^{k+1}\big),$$

and regrouping:

$$-r\,u_{j-1}^{k+1} + 2(1+r)u_j^{k+1} - r\,u_{j+1}^{k+1} = r\,u_{j-1}^k + 2(1-r)u_j^k + r\,u_{j+1}^k, \quad j=1,2,\ldots,n. \tag{20.13}$$

The formula's entries form a ⌶-shaped stencil on the finite difference grid. Two instances of the stencil are shown in Figure 20.5. If the stencil contacts the left or right boundary, as it has in one of the instances shown, it picks up the user-supplied boundary conditions there.

To fully grasp what the Crank–Nicolson scheme (20.13) represents, let us write it out in detail:

$$\begin{aligned}
j=1: &\qquad -r\,u_0^{k+1} + 2(1+r)u_1^{k+1} - r\,u_2^{k+1} = r\,u_0^k + 2(1-r)u_1^k + r\,u_2^k,\\
j=2: &\qquad -r\,u_1^{k+1} + 2(1+r)u_2^{k+1} - r\,u_3^{k+1} = r\,u_1^k + 2(1-r)u_2^k + r\,u_3^k,\\
\cdots &\qquad \cdots\\
j=n: &\qquad -r\,u_{n-1}^{k+1} + 2(1+r)u_n^{k+1} - r\,u_{n+1}^{k+1} = r\,u_{n-1}^k + 2(1-r)u_n^k + r\,u_{n+1}^k.
\end{aligned}$$

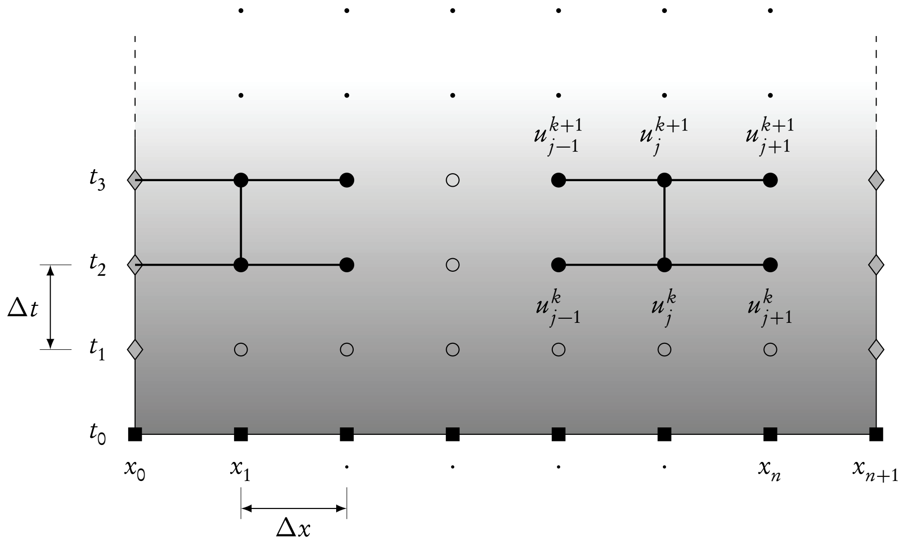


Figure 20.5: The Crank–Nicolson finite difference scheme acts on an ⌶-shaped stencil. It relates a linear combination of three values of u_j^{k+1} at the time-slice $k+1$ to three values of u from the previous time-slice. When the stencil hits the left or right boundary, its picks up the prescribed boundary values from there.

After setting[81] $s = 2(1+r)$ and $s' = 2(1-r)$, it takes the form

$$\begin{pmatrix} s & -r & & & \\ -r & s & -r & & \\ & \ddots & \ddots & \ddots & \\ & & -r & s & -r \\ & & & -r & s \end{pmatrix} \begin{pmatrix} u_1^{k+1} \\ u_2^{k+1} \\ \vdots \\ u_{n-1}^{k+1} \\ u_n^{k+1} \end{pmatrix} = \begin{pmatrix} s' & r & & & \\ r & s' & r & & \\ & \ddots & \ddots & \ddots & \\ & & r & s' & r \\ & & & r & s' \end{pmatrix} \begin{pmatrix} u_1^{k} \\ u_2^{k} \\ \vdots \\ u_{n-1}^{k} \\ u_n^{k} \end{pmatrix} + \begin{pmatrix} r\,u_0^k + r\,u_0^{k+1} \\ 0 \\ \vdots \\ 0 \\ r\,u_{n+1}^k + r\,u_{n+1}^{k+1} \end{pmatrix}. \tag{20.14}$$

It can be shown that the Crank–Nicolson scheme is unconditionally stable for all $r > 0$. That's good news. Even better news is that the discretization error of this scheme is of the order $(\Delta t)^2 + (\Delta x)^2$. Unlike the previous section's implicit scheme, here Δt and Δx occur in equal powers. This allows taking time-steps of the same order of magnitude as the space discretization. For this reason, Crank–Nicolson should be your default finite difference scheme unless other considerations prevail.

[81] Beware that $s = 2(1+r)$ is different from the previous definitions of s.

Computing the solution at the time-slice $k+1$ calls for solving the system of linear equations (20.14), which, as in the case of the previous section's implicit method, is tridiagonal and can be handled with the algorithm suggested in subsection 20.9.3.

20.5 ▪ The Seidman sweep scheme for the heat equation

In this section I introduce a lesser known difference scheme developed by Seidman [58] which has the dual advantages of being both *explicit* and *unconditionally stable*. As in the previous sections, I will explain the scheme in the context of the heat equation (20.2). The scheme's unique strength, however, is in the ease with which it may be implemented to solve nonlinear problems, as we shall see in Chapter 21.

I must add that the theory and analysis in [58] is developed in the context of general second order parabolic equations in n dimensions on irregular grids. What I present here is a very special case.

The *Seidman sweep*, as I shall call it, is very similar to the explicit scheme of Section 20.2 in that it approximates the time derivative by a forward difference. However, it calculates u_j^{k+1} in a sweeping motion, from left to right (forward sweep) or from right to left (reverse sweep), that is, in increasing or decreasing sequences of j. In the forward sweep, when processing node j, it takes advantage of the availability of u_{j-1}^{k+1} and uses it instead of u_{j-1}^{k}. In the reverse sweep it takes advantage of the availability of u_{j+1}^{k+1} and uses it instead of u_{j-1}^{k}. The idea is reminiscent of the *Gauss–Seidel* iterative scheme for solving linear systems of equations.

The algorithm splits the time-step Δt into two halves. A forward sweep advances time by $(\Delta t)/2$ and calculates $u_j^{k+1/2}$ from u_j^k, $j=1,2,\dots,n$. That is followed by a reverse sweep, which advances time by $(\Delta t)/2$ and calculates u_j^{k+1} from $u_j^{k+1/2}$, $j=n,\dots,2,1$. If we consider, for the moment, applying the explicit scheme of Section 20.2 to the forward and reverse sweeps, (20.6) will take the form

$$
\begin{aligned}
u_j^{k+1/2} - u_j^k &= r'(u_{j-1}^k - 2u_j^k + u_{j+1}^k), & j&=1,2,\dots,n,\\
u_j^{k+1} - u_j^{k+1/2} &= r'(u_{j-1}^{k+1/2} - 2u_j^{k+1/2} + u_{j+1}^{k+1/2}), & j&=n,\dots,2,1,
\end{aligned}
$$

where

$$
r' = \frac{r}{2} = \frac{\Delta t}{2(\Delta x)^2}, \tag{20.15}
$$

since we are taking half steps in time now.

I could present the rest of this section using the $k+1/2$ superscript notation, but the formulas become cumbersome and obscure the algorithm's simplicity. It works much better with a temporary convention where u_j, v_j, and w_j stand for the u_j^k, $u_j^{k+1/2}$, and u_j^{k+1}, respectively. With this notation, the pair of formulas shown above takes the form

$$
\begin{aligned}
v_j - u_j &= r'(u_{j-1} - 2u_j + u_{j+1}), & j&=1,2,\dots,n,\\
w_j - v_j &= r'(v_{j-1} - 2v_j + v_{j+1}), & j&=n,\dots,2,1,
\end{aligned}
$$

or equivalently,

$$
\begin{aligned}
v_j - u_j &= -r'(u_j - u_{j-1}) - r'(u_j - u_{j+1}), & j&=1,2,\dots,n,\\
w_j - v_j &= -r'(v_j - v_{j-1}) - r'(v_j - v_{j+1}), & j&=n,\dots,2,1.
\end{aligned}
$$

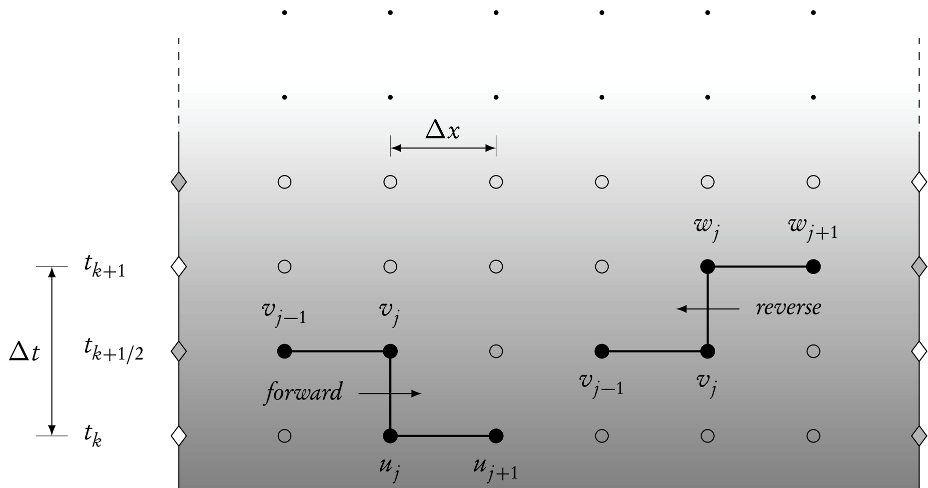


Figure 20.6: The Seidman sweep finite difference scheme advances from the time-slice k to the time-slice $k+1$ via an intermediate time-slice $k+1/2$. The forward sweep advances by half a time-step from k to $k+1/2$. The reverse sweep advances by half a time-step from $k+1/2$ to $k+1$. At any point it uses the most up-to-date data available. The boundary values at the points marked by the filled diamonds ◊ affect the solution in the interior points. The boundary values at the points marked by the hollow diamonds ◊ don't.

This is just the explicit scheme up to now. To change it over to the Seidman sweep, we note that in a *forward sweep* the value of v_{j-1} has been calculated prior to arriving at the node j. We take advantage of that and replace $(u_j - u_{j-1})$ in the first of the two formulas above by $(v_j - v_{j-1})$. Similarly, in a *reverse sweep* the value of w_{j+1} has been calculated prior to arriving at the node j. Therefore, we replace $(v_j - v_{j+1})$ in the second of the two formulas above by $(w_j - w_{j+1})$. These result in

$$v_j - u_j = -r'(v_j - v_{j-1}) - r'(u_j - u_{j+1}), \qquad j = 1, 2, \ldots, n, \tag{20.16a}$$

$$w_j - v_j = -r'(v_j - v_{j-1}) - r'(w_j - w_{j+1}), \qquad j = n, \ldots, 2, 1, \tag{20.16b}$$

which we rearrange into

$$(1 + r')v_j = r'v_{j-1} + (1 - r')u_j + r'u_{j+1}, \qquad j = 1, 2, \ldots, n, \tag{20.17a}$$

$$(1 + r')w_j = r'v_{j-1} + (1 - r')v_j + r'w_{j+1}, \qquad j = n, \ldots, 2, 1. \tag{20.17b}$$

The pair of formulas (20.17a) and (20.17b) constitutes the *Seidman sweep* scheme. It is an *explicit scheme* since all values on the right-hand sides are available at the time when the left-hand sides are evaluated. Figure 20.6 shows the stencils for the forward and reverse sweeps.

To express the Seidman sweep as a matrix-vector equation, it is best to use the (20.16) form of the scheme. Upon inspection of that formula we see that

$$
\begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_{n-1} \\ v_n \end{pmatrix} - \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_{n-1} \\ u_n \end{pmatrix} = -r' \begin{pmatrix} 1 & 0 & & & \\ -1 & 1 & 0 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 1 & 0 \\ & & & -1 & 1 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_{n-1} \\ v_n \end{pmatrix}
$$

$$
- r' \begin{pmatrix} 1 & -1 & & & \\ 0 & 1 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & 0 & 1 & -1 \\ & & & 0 & 1 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_{n-1} \\ u_n \end{pmatrix} + r' \begin{pmatrix} v_0 \\ 0 \\ \vdots \\ 0 \\ u_{n+1} \end{pmatrix}.
$$

Writing $\boldsymbol{u}$, $\boldsymbol{v}$, and $\boldsymbol{w}$ for the column vectors with the components u_j, v_j, w_j and letting

$$
A = \begin{pmatrix} 1 & 0 & & & \\ -1 & 1 & 0 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 1 & 0 \\ & & & -1 & 1 \end{pmatrix}, \quad \boldsymbol{a} = \begin{pmatrix} v_0 \\ 0 \\ \vdots \\ 0 \\ u_{n+1} \end{pmatrix}, \quad \boldsymbol{b} = \begin{pmatrix} v_0 \\ 0 \\ \vdots \\ 0 \\ w_{n+1} \end{pmatrix},
$$

this takes on the compact form $\boldsymbol{v} - \boldsymbol{u} = -r'A\boldsymbol{v} - r'A^T\boldsymbol{u} + r'\boldsymbol{a}$, or equivalently,

$$
(I + r'A)\boldsymbol{v} = (I - r'A^T)\boldsymbol{u} + r'\boldsymbol{a}, \tag{20.18a}
$$

where I is the identity matrix and A^T is the transpose of A. Similar considerations regarding the reverse sweep in the formula (20.16b) lead to

$$
(I + r'A^T)\boldsymbol{w} = (I - r'A)\boldsymbol{v} + r'\boldsymbol{b}. \tag{20.18b}
$$

The analysis in [58] shows that the iteration scheme expressed in the pair of equations (20.18a) and (20.18b) is *unconditionally stable* for all $r' > 0$. I am not aware of a study of the scheme's *rate of convergence*, and I have not analyzed it myself. Numerical experiments—see the error graphs in Figure 20.7—point to a rate of convergence of the order $(\Delta x)^2 + (\Delta t)^2 + \frac{(\Delta t)^2}{(\Delta x)^2}$, like that of the Du Fort–Frankel scheme (see [71]), but that's a mere conjecture on my part.

In favor of the Seidman sweep, one may note the following:

1. The Seidman sweep, being an explicit method, is easy to program, as we shall see later in this chapter.
2. The Seidman sweep handles discontinuities in the initial and boundary conditions more gracefully than Crank–Nicolson.
3. The Seidman sweep has a definite advantage over implicit schemes in solving nonlinear problems. An implicit scheme, such as Crank–Nicolson, requires solving an $n \times n$ nonlinear tridiagonal system at every step. The Seidman sweep, being an explicit scheme, needs to solve $2n$ single (uncoupled) nonlinear equations in every forward/reverse sweep pair. Chapter 21 applies the Seidman sweep to solve the highly nonlinear porous medium equation.

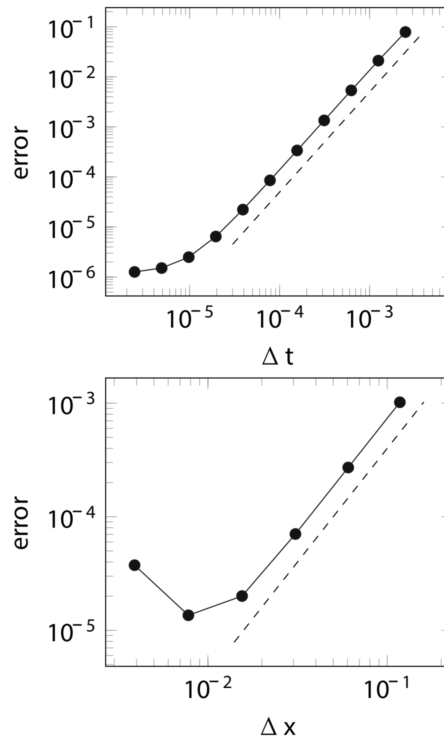


Figure 20.7: Experiments with the Seidman sweep and *problem heat1* (Section 20.6) lend support to the conjecture that the scheme's convergence rate may be $O\big((\Delta x)^2 + (\Delta t)^2 + \frac{(\Delta t)^2}{(\Delta x)^2}\big)$. On the top we have the graph of the errors versus Δt while $\Delta x = 0.004$ is kept fixed. On the bottom we have the graph of the errors versus Δx while $\Delta t = 5 \times 10^{-5}$ is kept fixed. The dashed lines have slopes of 2. The upturned tail in the latter graph is characteristic of the presence of a $\frac{\Delta t}{\Delta x}$ term.

20.6 ▪ Test problems

Here I introduce four simple initial/boundary value problems for the purpose of testing and demonstrating the various finite difference discretization schemes that were introduced in the previous sections. The programs which we are going to develop are general and certainly not limited to these four. You may easily modify those problems or add new ones of your own. The partial differential equation in all four problems is the heat equation (20.1) on the interval $-1 < x < 1$. Only the initial and boundary conditions are different.

The interval $-1 < x < 1$ is not hard-coded anywhere. The programs are set up to solve a finite difference problem on an interval $a < x < b$, where a and b are defined alongside the rest of the problem's data. Don't hesitate to experiment with defining and solving problems on intervals other than $-1 < x < 1$.

Problems *heat1* and *heat2* come with exact solutions. Our programs compare these against the finite difference solutions and print out the discretization errors. These two problems are constructed according to the following simple "reverse engineering" idea.

Pick any function, let's say $u_{\text{ex}}(x,t)$, that satisfies the partial differential equation (20.1) for all $-\infty < x < \infty$ and $t > 0$. Pose the following initial/boundary value problem whose

data is defined in terms of $u_{\text{ex}}(x,t)$:

$$\begin{cases} \dfrac{\partial u}{\partial t} = \dfrac{\partial^2 u}{\partial x^2}, & x \in (a,b),\ t > 0, \\ u(x,0) = u_{\text{ex}}(x,0), & x \in (a,b), \\ u(a,t) = u_{\text{ex}}(a,t), & t > 0, \\ u(b,t) = u_{\text{ex}}(b,t), & t > 0. \end{cases} \tag{20.19}$$

Clearly $u(x,t) = u_{\text{ex}}(x,t)$ is a solution of the problem. (Actually, it's *the* solution since such problems have unique solutions.) To test the accuracy of our solvers, we have them solve (20.19) and then compare the results with $u_{\text{ex}}(x,t)$.

Problem *heat1*: You should have no difficulty in verifying that the function

$$u_{\text{ex}}(x,t) = e^{-\frac{1}{2}\pi^2 t} \cos \frac{1}{2}\pi x \tag{20.20}$$

is a solution of the heat equation (20.1). Check that for yourself! We define the problem *heat1* by plugging (20.20) into the general template (20.19) with $a = -1$, $b = 1$.

Problem *heat2*: The *error function*, erf, despite its infelicitous appellation, occurs quite frequently in the study of differential equations, probability, and statistics. It is defined as

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, dt.$$

The domain of erf is the entire real line. It increases monotonically from -1 at $x = -\infty$ to $+1$ at $x = +\infty$. Figure 20.8 depicts its graph. Since the derivative of $\operatorname{erf}(x)$ is $\frac{2}{\sqrt{\pi}} e^{-x^2}$, you should be able to verify without much trouble that the function

$$u_{\text{ex}}(x,t) = \operatorname{erf}\left(\frac{x}{2\sqrt{t}}\right) \tag{20.21}$$

is an exact solution of the heat equation (20.1). We wish to use this finding to "reverse-engineer" an initial/boundary value problem according to the template (20.19), but we hit a snag: The second of the equations in (20.1) calls for the value of $u_{\text{ex}}(x,0)$. Plugging $t = 0$ in (20.21) won't do since that would entail a division by zero on account of the $\sqrt{t}$ in the denominator. We get around this by observing that although the expression $u_{\text{ex}}(x,t)$ is undefined at $t = 0$, its limit as t approaches zero from above *does* exist:

$$\lim_{t \to 0^+} u_{\text{ex}}(x,t) = \begin{cases} -1 & \text{if } x < 0, \\ +1 & \text{if } x > 0. \end{cases}$$

This still leaves out the $x = 0$ case. There is no way around that since $u_{\text{ex}}(x,t)$ is irreparably discontinuous there. That's not a significant obstacle, however, since the heat equation is good at smearing out discontinuities. Any value assigned to $u_{\text{ex}}(0,0)$ will fade away quite fast. In our program we let $u_{\text{ex}}(0,0) = 0$. That's as good a choice as any. Thus, our u_{ex} really looks like this:

$$u_{\text{ex}}(x,t) = \begin{cases} \operatorname{erf}\left(\frac{x}{2\sqrt{t}}\right) & \text{if } t > 0, \\ -1 & \text{if } t = 0 \text{ and } x < 0, \\ +1 & \text{if } t = 0 \text{ and } x > 0, \\ 0 & \text{if } t = 0 \text{ and } x = 0. \end{cases} \tag{20.22}$$

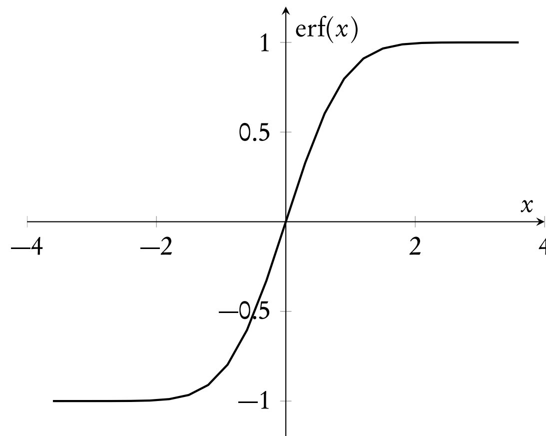


Figure 20.8: The graph of the *error function* erf(x).

With the continuity issue out of the way, we define the problem *heat2* by plugging (20.22) into the general template (20.19) with $a = -1$, $b = 1$.

Remark 20.1. Due to the linearity of the heat equation (20.1), any constant multiple of a solution is also a solution. In the implementation of problem *heat2*, I use $0.4 \times u_{\text{ex}}(x,t)$ since the resulting flattened graph has a more appealing look. There is no deep reason behind that choice.

Problem *heat3*: Take the "rectangular bump" function

$$u_0(x) = \begin{cases} 1, & |x| < 0.4, \\ 0 & \text{otherwise} \end{cases}$$

for the initial condition, and define the problem *heat3* as

$$\begin{cases} \dfrac{\partial u}{\partial t} = \dfrac{\partial^2 u}{\partial x^2}, & x \in (-1,1),\ t > 0, \\ u(x,0) = u_0(x), & x \in (-1,1), \\ u(-1,t) = u(1,t) = 0, & t > 0. \end{cases}$$

Problem *heat4*: This problem is defined as

$$\begin{cases} \dfrac{\partial u}{\partial t} = \dfrac{\partial^2 u}{\partial x^2}, & x \in (-1,1),\ t > 0, \\ u(x,0) = 0, & x \in (-1,1), \\ u(-1,t) = 0, & t > 0, \\ u(1,t) = 0.8, & t > 0. \end{cases}$$

What does this say about the value of the solution at $x = 1$, $t = 0$?. The initial condition says $u(1,0)$ is 0. The boundary condition says $u(1,0)$ is 0.8. Therefore we expect the solution $u(x,t)$ to be discontinuous at $x = 0$, $t = 0$. The purpose of this problem is to examine how well the various finite difference schemes handle that discontinuity.

20.7 ▪ The program

The rest of this chapter is devoted to details of the implementations of the four finite difference schemes introduced in the previous sections. We will write four individual programs, one for each scheme, encoded in the files *heat-explicit.c*, *heat-implicit.c*, *heat-crank-nicolson.c*, and *heat-seidman-sweep.c*. The problems to be solved are defined in a file named *problem-spec.c*, which may contain any number of initial/boundary value problems.

We rely on *xmalloc.[ch]* from Chapter 7 to allocate memory and the file *array.h* from Chapter 8 to construct vectors and matrices. Thus, following the suggestions in Chapters 2 and 6, the contents of the project's directory will look like this:

```
$ cd fd1
$ ls -F
Makefile               heat-implicit.c       xmalloc.c@
array.h@               heat-seidman-sweep.c  xmalloc.h@
heat-crank-nicolson.c  problem-spec.c
heat-explicit.c        problem-spec.h
```

I have named my directory `fd1` since the programs here deal with finite difference schemes in a one-dimensional space.

There is a good deal of duplication among the *heat-*.c* files since the supporting code—parsing the command-line arguments, computing errors, plotting the solutions—is common to all four. It is possible to merge the four programs into one and factor out the common parts, but the logic can get a bit twisty; therefore I will not go down that path. You are welcome to try it if you wish.

Currently my *problem-spec.c* contains four boundary value problems named *heat1*, *heat2*, *heat3*, *heat4* that were introduced in the previous section. Each of these can be solved with any of the four finite difference schemes; therefore we are going to produce a total of 16 solutions altogether.

Conceptually, each solution is a matrix of the computed u_j^k values of the solution at the finite difference grid points. We may set up the programs to print out those matrices as tables of numbers, but it is more informative to produce pictures of the graphs of the solutions $u(x,t)$ as surfaces drawn in three dimensions. Thus, for each solution we write the grid values to a file in a special format that is suitable for feeding to *Geomview* (see page 8) which will render the surface on your computer screen and let you rotate it with the mouse. The graphs of the various solutions shown in this chapter are snapshots of a *Geomview* window.

Here is a transcript of a sample session with the *heat-explicit* program that implements the explicit scheme:

```
$ ./heat-explicit
Usage: ./heat-explicit T n s
   T : solve over 0 ≤ t ≤ T
   n : number of grid points a=x[0], x[1], ... , x[n], x[n+1]=b
   s : number of time-slices 0=t[0], t[1], ... , t[s]=T

$ ./heat-explicit 0.2 9 10
problem heat1:
-1 < x < 1,  0 < t < 0.2,  dx = 0.2, dt = 0.02,  r = dt/dx^2 = 0.5
geomview script written to file ex1.gv
max error at time 0.2 is 0.00506898
```

```
problem heat2:
-1 < x < 1,  0 < t < 0.2,  dx = 0.2, dt = 0.02,  r = dt/dx^2 = 0.5
geomview script written to file ex2.gv
max error at time 0.2 is 0.00999372

problem heat3:
-1 < x < 1,  0 < t < 0.2,  dx = 0.2, dt = 0.02,  r = dt/dx^2 = 0.5
geomview script written to file ex3.gv

problem heat4:
-1 < x < 1,  0 < t < 0.2,  dx = 0.2, dt = 0.02,  r = dt/dx^2 = 0.5
geomview script written to file ex4.gv
```

The meanings of the program's arguments should be clear from the "Usage" message that appears above. The programs *heat-implicit*, *heat-crank-nicolson.c*, and *heat-seidman-sweep.c* may be invoked in the same way. Figures 20.9 and 20.10 shows the results of running the four programs, each solving the four problems *heat1*, *heat2*, *heat3*, and *heat4*. The figures' captions give the command-line arguments with which the programs were invoked.

Since the problems *heat1* and *heat2* are supplied with exact solutions, the program computes and prints the error in those two cases. The error is defined by

$$\text{err} = \max_{0 \le j \le n+1} |u_{\text{ex}}(x_j, T) - u_j^s|, \tag{20.23}$$

where the superscript s corresponds to the time index at the time T, that is, $t_s = T$.

Let us note that *heat-explicit* was involved with the command-line arguments, 0.5 10 32. In words, the problem is solved over the range $0 \le t \le 0.5$ through 32 time-steps; therefore $\Delta t = 0.5/32 = 1/64$. The x domain, which is the interval $(-1, 1)$, has 10 internal nodes; therefore $\Delta x = 2/11$. We see that $r = (\Delta t)/(\Delta x)^2 \approx 0.472656 < 0.5$, that is, the scheme is within the threshold of stability; see Section 20.2. In contrast, if we run *heat-explicit* with the arguments 0.352 20 20, we will have $r = 1.9404 > 0.5$; therefore the scheme will be unstable. Figure 20.11 shows the resulting calamity when it's applied to the problem *heat1*.

20.8 ▪ The files *problem-spec.[ch]*

The file *problem-spec.h* provides a data structure designed to hold the definition of an initial/boundary value problem. The file *problem-spec.c* uses that data structure to define concrete realizations of any number of such problems. The next two subsections give the details of the contents of these files.

20.8.1 ▪ The file *problem-spec.h*

The file *problem-spec.h*, shown in it entirety in Listing 20.1, declares a data structure named "**`struct`** `problem_spec`" for storing definitions of initial/boundary value problems of the type (20.2) on page 251. The members `a` and `b` of that structure hold the coordinates of the endpoints of the interval $a \le x \le b$. The member `ic` points to a function that supplies the problem's initial condition $u_0(x)$, while the members `bcL` and `bcR` point to functions $u_L(t)$ and $u_R(t)$ that supply the left and right boundary conditions. Finally, the member `u_exact` points to a function that returns the problem's exact solution. If such a function is provided, the program will use it to compute and print the error (at the final time) in the finite difference scheme. If an exact solution is not available, set that pointer to `NULL`.

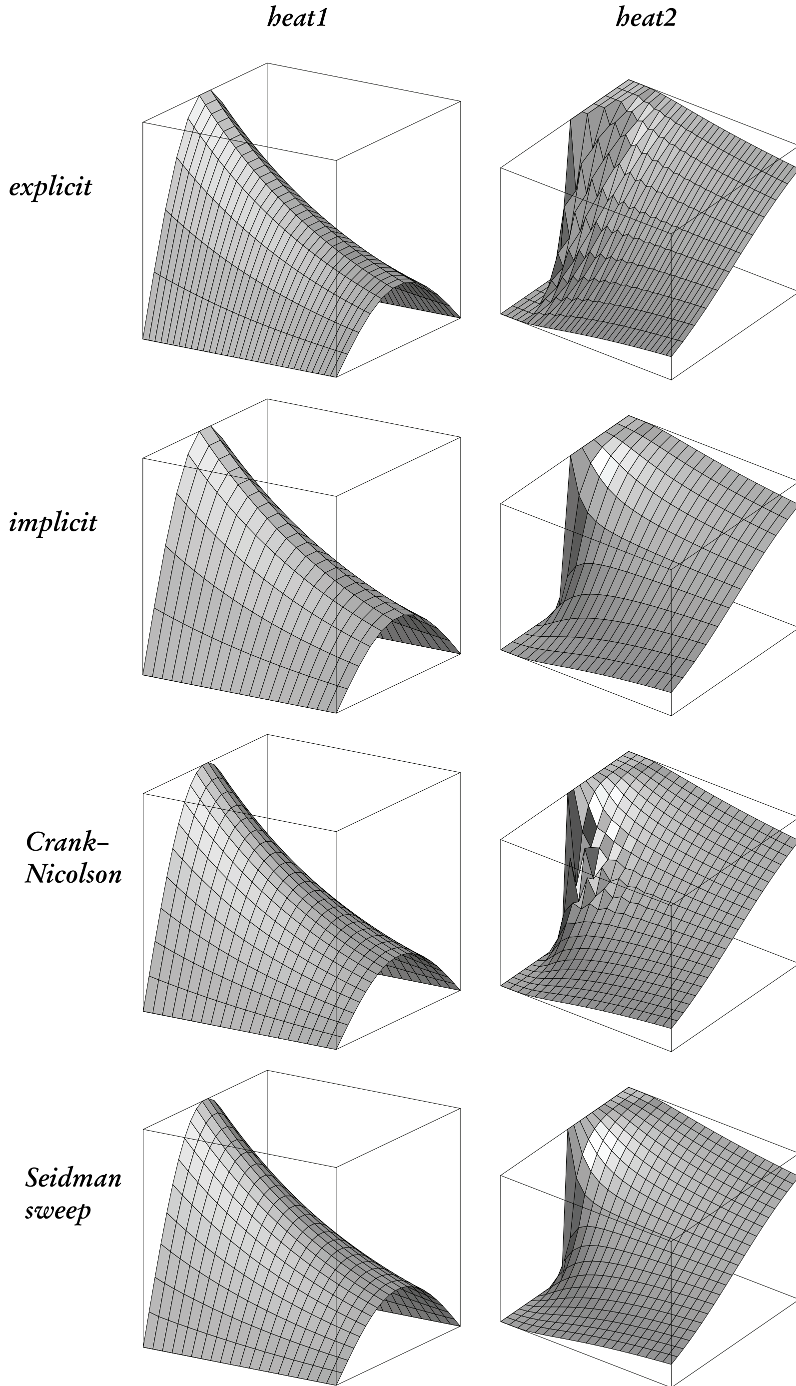


Figure 20.9: Graphs of the solutions to the problems *heat1* and *heat2* produced by the four schemes invoked as `heat-explicit 0.5 10 32`, `heat-implicit 0.5 10 20`, `heat-crank-nicolson 0.5 20 20`, `heat-seidman-sweep 0.5 20 20`.

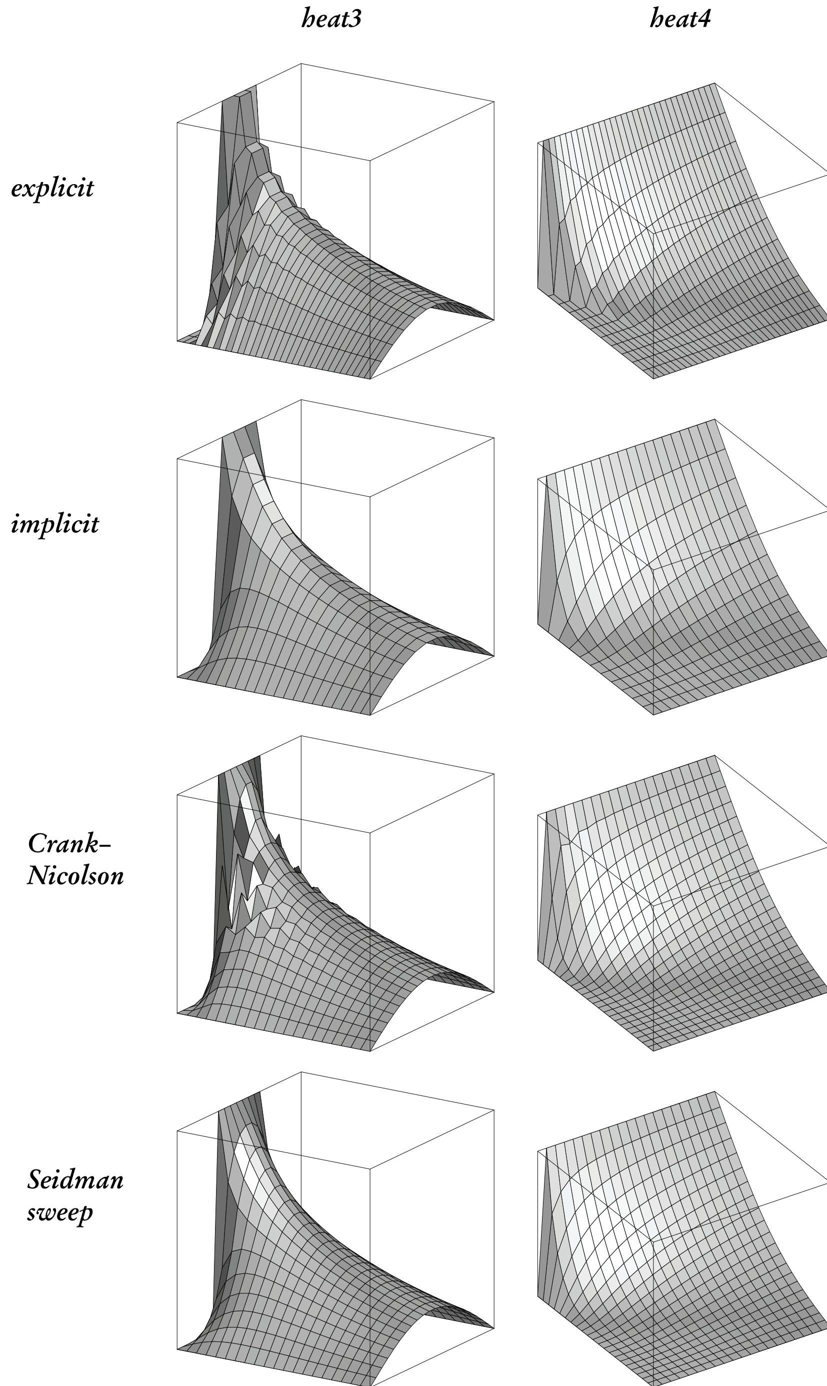


Figure 20.10: Graphs of the solutions to the problems *heat3* and *heat4* produced by the four schemes invoked as `heat-explicit 0.5 10 32`, `heat-implicit 0.5 10 20`, `heat-crank-nicolson 0.5 20 20`, `heat-seidman-sweep 0.5 20 20`.

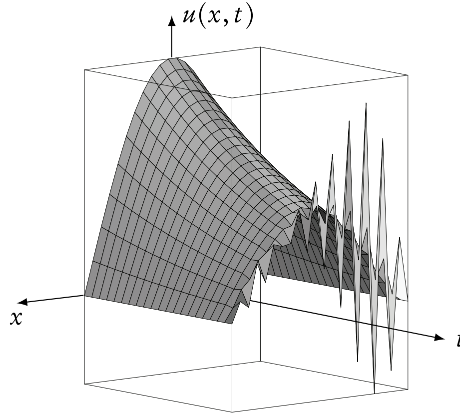


Figure 20.11: Running *heat-explicit* with the command-line arguments `0.352 20 20` yields an unstable scheme since $r = (\Delta t)/(\Delta x)^2 \approx 1.9404 > 0.5$. Here we see the solution of the problem *heat1* begin to fall apart as we approach $t = 0.352$. Beyond this the oscillations grow very large so quickly that drawing a graph is impractical.

Listing 20.1: The contents of the file *problem-spec.h*.

```
1  #ifndef H_PROBLEM_SPEC_H
2  #define H_PROBLEM_SPEC_H
3  struct problem_spec {
4          double a;                                // left end at x = a
5          double b;                                // right end at x= b
6          double (*ic)(double x);                  // initial condition
7          double (*bcL)(double t);                 // left boundary condition
8          double (*bcR)(double t);                 // right boundary condition
9          double (*u_exact)(double x, double t);   // exact solution, if any
10 };
11 #endif /* H_PROBLEM_SPEC_H */
```

20.8.2 ▪ The file *problem-spec.c*

My *problem-spec.c* contains the definitions of the four initial/boundary value problems *heat1*, *heat2*, *heat3*, and *heat4* introduced in Section 20.6. You are invited to modify these or add problems of your own.

Listing 20.2 shows the part of the file *problem-spec.c* where the problem named *heat1* is defined. Following the "reverse-engineering" idea expounded in (20.19), I use the exact solution given in (20.20) to set the problem's initial and boundary conditions. Lines 6 through 22 are self-explanatory, so there is no need to elaborate other than note that the cosine function, `cos()`, and the exponential function, `exp()`, are defined in C's standard mathematics library; therefore you may need to link it to your program by specifying the `-lm` flag at the linking stage. The header file *math.h* (line 2) provides the prototypes of those functions.

Lines 23 through 35 define a function named `heat1()` that sets up and populates a **`struct`** `problem_spec` with the problem's data and returns the structure's address. The **`static`** specifier on line 25 is *absolutely essential*. A static object is available

Listing 20.2: This is the top of my file *problem-spec.c*. It shows the requisite **#include**s, plus all the components that define the problem *heat1*. I use the exact solution given in (20.20) to set the problem's initial and boundary conditions.

```
1  #include <stdio.h>
2  #include <math.h>
3  #include "problem-spec.h"
4
5  // heat1(): quite a basic one-dimensional heat equation with exact solution
6  static double heat1_exact(double x, double t)
7  {
8      double Pi = 4*atan(1);
9      return exp(-Pi*Pi/4*t) * cos(Pi/2*x);
10 }
11 static double heat1_ic(double x)
12 {
13     return heat1_exact(x,0);
14 }
15 static double heat1_bcL(double t)
16 {
17     return heat1_exact(-1,t);
18 }
19 static double heat1_bcR(double t)
20 {
21     return heat1_exact(1,t);
22 }
23 struct problem_spec *heat1(void)
24 {
25     static struct problem_spec spec = {   // C99-style initialization!
26         .a          = -1.0,
27         .b          =  1.0,
28         .ic         = heat1_ic,
29         .bcL        = heat1_bcL,
30         .bcR        = heat1_bcR,
31         .u_exact    = heat1_exact,
32     };
33     printf("problem heat1:\n");
34     return &spec;
35 }
```

throughout the duration of the program's execution. Without the **static** specifier, the structure will come into existence when the function is entered and will evaporate when the function is exited. In that case returning the structure's address would make no sense at all since it would be the address of an evaporated entity.[82]

I am initializing the structure *à la* C99. The program won't compile if your compiler does not recognize the C99 syntax. See the comments under the **Initializing structures** heading on page 6 if you wish to change the initialization to the equivalent C89 syntax.

[82]Listing 20.2 exhibits two *completely unrelated* meanings of C's **static** keyword which should not be confused. The one on line 25 pertains to the corresponding object's *lifetime*, as explained above. All other occurrences of **static** in that listing pertain to *linkage*. The one on line 6, for instance, indicates that the function `heat1_exact()` is inaccessible/invisible outside this file. Consult your C reference/manual for a more detailed explanation.

Listing 20.3: An outline of the file *heat-implicit.c*.

```
1  ▶ #include ...
2  ▶ static void trisolve(int n, double *a, double *d, double *c, double *b,
3            double *x) ...
4  ▶ static double get_error(struct problem_spec *spec,
5            double *u, int n, double T) ...
6  ▶ static void plot_curve(FILE *fp, double *u, int n, int steps, int k) ...
7  ▶ static void heat_implicit(struct problem_spec *spec,
8            double T, int n, int steps, char *gv_filename) ...
9  ▶ static void show_usage(char *progname) ...
10 ▶ int main(int argc, char **argv) ...
```

The problem *heat2* is defined in the same way. We use the exact solution (20.22) to set the problem's initial and boundary conditions. I will leave it to you to supply the details. You will need to know that the error function, erf, is defined in C's standard mathematics library and is called `erf()`.

Problems *heat3* and *heat4* do not come with predefined exact solutions; therefore we define their initial and boundary conditions from scratch. For example,

```
1 static double heat3_ic(double x)
2 {
3     return fabs(x) < 0.4 ? 1.0 : 0.0;
4 }
```

Again, I will leave it to you to supply the remaining details.

20.9 ▪ The file *heat-implicit.c*

In this section I will go through the complete contents of the file *heat-implicit.c*, which provides an implementation of the implicit finite difference scheme for solving the heat equation. I will leave the writing of the files *heat-explicit.c*, *heat-crank-nicolson.c*, and *heat-seidman-sweep.c* as projects for you.

Listing 20.3 provides an outline of *heat-implicit.c*. The function `main()` parses the command-line arguments and then makes a sequence of calls, one per problem to be solved, to `heat_implicit()`, which, in turn, sets up a **`for`**-loop to march forward through the time steps and at each step calls `trisolve()` to solve the tridiagonal system (20.12) and `plot_curve()` to construct a section of the three-dimensional surface plots that we see in Figures 20.9 and 20.10. Upon exiting the loop, if an exact solution is provided, the function `get_error()` is called to print the discrepancy between the exact and computed solutions. The details of these functions are described in the subsections below, albeit not in the order that they appear in Listing 20.3.

20.9.1 ▪ The function `main()`

As we saw in the transcript of the interactive session with the program on page 266, the program is expected to be invoked with three arguments, as in

```
$ ./heat-implicit T n s
```

where T specifies the upper end of the time range $0 \le t \le T$, n is the number of the *internal* grid points in the x direction, and s is the number of time-steps. To be precise,

Listing 20.4: The function `main()` in the file *heat-implicit.c*. Lines 3–6 supply the prototypes of the functions `heat1()` through `heat4()`, which are defined in *problem-spec.c*.

```
1  int main(int argc, char **argv)
2  {
3      struct problem_spec *heat1(void);
4      struct problem_spec *heat2(void);
5      struct problem_spec *heat3(void);
6      struct problem_spec *heat4(void);
7      char *endptr;
8      double T;
9      int n, steps;
10     if (argc ≠ 4) {
11         show_usage(argv[0]);
12         return EXIT_FAILURE;
13     }
14     T = strtod(argv[1], &endptr);
15     if (*endptr ≠ '\0' || T ≤ 0.0) {
16         show_usage(argv[0]);
17         return EXIT_FAILURE;
18     }
19     n = strtol(argv[2], &endptr, 10);
20     if (*endptr ≠ '\0' || n < 1) {
21         show_usage(argv[0]);
22         return EXIT_FAILURE;
23     }
24     steps = strtol(argv[3], &endptr, 10);
25     if (*endptr ≠ '\0' || steps < 0) {
26         show_usage(argv[0]);
27         return EXIT_FAILURE;
28     }
29     heat_implicit(heat1(), T, n, steps, "im1.gv");
30     heat_implicit(heat2(), T, n, steps, "im2.gv");
31     heat_implicit(heat3(), T, n, steps, "im3.gv");
32     heat_implicit(heat4(), T, n, steps, "im4.gv");
33     return EXIT_SUCCESS;
34 }
```

the x coordinates of the grid points are $a = x_0, x_1, \dots, x_n, x_{n+1} = b$, and the time-slices take place at $0 = t_0, t_1, \dots, t_s = T$. Therefore $\Delta t = T/s$ and $\Delta x = (b-a)/(n+1)$.

One of the tasks of the function `main()` that appears on line 10 of Listing 20.3 is to extract the values of `T`, `n`, and `s` from the command-line. Listing 20.4 gives the details of its implementations.

Lines 3–6 supply the prototypes of the functions `heat1()` through `heat4()`, which are defined in *problem-spec.c*. I could have put those prototypes in the file *problem-spec.h* but chose not to since I don't like the idea of a *problem-spec.h* file which changes with the addition of every new problem in *problem-spec.c*. I have placed those prototypes in `main()` since that's the only place where they are needed.

Lines 10–28 extract the values of T, n, and s (the last is called `steps` in the code) from the command-line. See Chapter 5 regarding the functions `strtod()` and `strtol()`. The next four lines call the function `heat_implicit()` four times to solve the four problems *heat1*, *heat2*, *heat3*, and *heat4*. The arguments `heat1()`, `heat2()`, `heat3()`,

and `heat4()` retrieve the "`struct problem_spec`" for each of the four problems defined in *problem-spec.c*. The arguments `"im1.gv"`, etc., are file names to which the program is to write the computed solutions in the form of *Geomview* scripts for plotting. The file names are arbitrary, but it's a good idea to name them so that they are easy for you to identify. When solving the problems with the Crank–Nicolson method, for instance, I would name them `"cn1.gv"`, etc.

20.9.2 ▪ The function `show_usage()`

The function `show_usage()` that appears on line 9 of Listing 20.3 is responsible for printing the "Usage" message shown in the transcript of the interactive session on page 266. It is called in several places in Listing 20.4. Implement the function, and add it to your *heat-implicit.c*.

20.9.3 ▪ The function `trisolve()`

The function `trisolve()` that appears on line 2 of Listing 20.3 is a generic solver of $n \times n$ tridiagonal linear systems of equations of the form

$$\begin{pmatrix} d_0 & c_0 & & & & \\ a_0 & d_1 & c_1 & & & \\ & a_1 & d_2 & c_2 & & \\ & & \ddots & \ddots & \ddots & \\ & & & a_{n-3} & d_{n-2} & c_{n-2} \\ & & & & a_{n-2} & d_{n-1} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{n-2} \\ x_{n-1} \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{n-2} \\ b_{n-1} \end{pmatrix}.$$

There is no point in storing the full $n \times n$ coefficient matrix. We store only the diagonal $\langle d_0, \dots, d_{n-1} \rangle$, the subdiagonal $\langle a_0, \dots, a_{n-2} \rangle$, and the superdiagonal $\langle c_0, \dots, c_{n-2} \rangle$ as ordinary vectors.

We assume that the coefficient matrix is nonsingular. Our code does not check for that. It can be shown that matrices that arise through finite difference schemes applied to parabolic problems are nonsingular.

We solve the system through a simple Gaussian elimination without pivoting. To eliminate a_0, we introduce the multiplier $m = a_0/d_0$ and then subtract m times the first equation from the second equation. This (i) reduces the entry in the a_0 position to zero; (ii) changes the entry in the d_1 position to $d_1 - mc_0$; and (iii) changes the entry in the b_1 position to $b_1 - mb_0$. The entry in the c_1 position does not change.

We repeat the procedure to eliminate the rest of the subdiagonal, in sequence, from top to bottom. When operating on row i ($i = 1, 2, \dots, n-1$) we set $m = a_{i-1}/d_{i-1}$ and then subtract m times row $i-1$ from row i. This (i) eliminates the a_{i-1} entry; (ii) changes the d_i entry to $d_i - mc_{i-1}$; and (iii) changes the b_i entry to $b_i - mb_{i-1}$.

Listing 20.5 shows my implementation of `trisolve()`. In lines 4–8 you will find the literal encoding of the statements made above.

Remark 20.2. We don't bother to zero the a_i entries in our code since we have no use for the a_i's beyond this point. Consequently, the elimination's overall effect is to change the d_i and b_i vectors, but the a_i and c_i vectors remain unchanged! We take advantage of this when calling `trisolve()`, as we will see shortly.

Listing 20.5: The function `trisolve()` in the file *heat-implicit.c*.

```
1  static void trisolve(int n, double *a, double *d, double *c, double *b,
2          double *x)
3  {
4      for (int i = 1; i < n; i++) {
5          double m = a[i-1]/d[i-1];
6          d[i] -= m*c[i-1];
7          b[i] -= m*b[i-1];
8      }
9      x[n-1] = b[n-1]/d[n-1];
10     for (int i = n-2; i ≥ 0; i--)
11         x[i] = (b[i] - c[i]*x[i+1]) / d[i];
12 }
```

After eliminating the subdiagonal, the system takes the form

$$\begin{pmatrix} d_0 & c_0 & & & & \\ & d_1 & c_1 & & & \\ & & d_2 & c_2 & & \\ & & & \ddots & \ddots & \\ & & & & d_{n-2} & c_{n-2} \\ & & & & & d_{n-1} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{n-2} \\ x_{n-1} \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{n-2} \\ b_{n-1} \end{pmatrix}.$$

The d_i's and b_i's are different from what they were before, but the c_i's are the same. The system's last equation, that is, $d_{n-1}x_{n-1} = b_{n-1}$, yields $x_{n-1} = b_{n-1}/d_{n-1}$. We compute the rest of the x_i's through *back substitution*. To wit, the equation before the last, that is, $d_{n-2}x_{n-2} + c_{n-2}x_{n-1} = b_{n-2}$, yields the value of x_{n-2} since x_{n-1} is already known. In general, the equation in row i ($i = n-2, \dots, 0$), that is, $d_i x_i + c_i x_{i+1} = b_i$, yields $x_i = (b_i - c_i x_{i+1})/d_i$ in terms of the previously calculated x_{i+1}. Lines 9–11 in Listing 20.5 reflect this finding.

20.9.4 ▪ The function `plot_curve()`

The function `plot_curve()` that appears on line 6 of Listing 20.3 is responsible for producing the graphs shown in Figures 20.9 and 20.10 on pages 268 and 269. Well, to be pedantic, `plot_curve()` does not *literally* produce those graphs, *Geomview* does. What `plot_curve()` does is to help write a script which when fed to *Geomview* produces those graphs.

To understand the details, we need to know what *Geomview* expects to see in its input script. The requirement is quite simple—to plot the graph of a function $z = f(x, y)$ over an $m \times n$ mesh, we supply the graph's (x, y, z) coordinates at the mesh points, where the mesh is scanned from left to right in each row and then by rows from top to bottom. We wrap the resulting sequence of $N = mn$ coordinate triplets (x, y, z) in a minimal code to make it palatable to *Geomview* and save it to a file, like this:

```
# the skeleton of a Geomview input file
{ appearance { +edge }
MESH m n
x[1] y[1] z[1]
x[2] y[2] z[2]
```

```
...
x[N] y[N] z[N]
}
```

The numbers are separated by *whitespace*.[83] One or more consecutive whitespace count as one whitespace. Line breaks are not required; you may put the $3N$ numbers all on a single line if you like. The "`appearance { +edge }`" at the top of the script tells *Geomview* to draw the mesh lines. If you change `+edge` to `-edge`, the surface will be drawn without the mesh lines. The keyword "`MESH`" should be typed in capital letters, as shown.

This is exactly how the graphs in Figures 20.9 and 20.10 were produced. Each graph depicts the solution u_j^k on an $(n+2)\times(s+1)$ grid, where $n+2$ and $s+1$ are the numbers of grid points in the x and t directions, respectively. As we will see in the next subsection, the values of u_j^k are computed iteratively, one time-slice at a time, as vectors

$$\boldsymbol{u} = \langle u_0^k, u_1^k, \dots, u_n^k, u_{n+1}^k \rangle \tag{20.24}$$

of length $n+2$. The function `plot_curve()` receives the vector $\boldsymbol{u}$. It calculates the coordinates x_j of the entry u_j^k and appends the $n+2$ triplets (x_j, t_k, u_j^k) to the *Geomview* script. Since `plot_curve()` handles only one time-slice of the overall graph, it is named "*plot curve*" rather than "*plot surface*".

Calculating the coordinates x_j is straightforward. The finite difference scheme has n internal grid points in the x direction; therefore the interval $a \le x \le b$ is divided into $n+1$ subintervals of length $(b-a)/(n+1)$ each. It follows that the jth grid point is at $x_j = a + \big[(b-a)/(n+1)\big]j$, $j = 0, 1, \dots, n+1$. Similarly, since we are taking s steps over the time interval $0 \le t \le T$, then $t_k = Tk/s$, $k = 0, 1, \dots, s$.

A direct implementation of the calculation above, however, can produce disappointing results. Here is why.

Suppose we solve an initial/boundary value problem over the domain $D = \{(x,t) : -1 \le x \le 1,\ 0 \le t \le 0.1\}$, which is a 2×0.1 rectangle. Plotting the solution to the correct scale over this long and narrow rectangle will produce a pretty useless graph. Since it's so narrow, one won't be able to see any details.

It is preferable to forgo the correct scaling in favor of stretching D into a square for plotting purposes. That's quite a natural thing to do, and that is what I have done in all the graphs shown in this chapter. Had you noticed that before I mentioned it?

So here is the idea: Instead of using the true x_j coordinates calculated above, just use $x_j = j/(n+1)$, $j = 0, 1, \dots, n+1$. As j ranges from 0 to $n+1$, the values of x_j range from 0 to 1. Similarly, instead of using the true t_k values calculated above, just take $t_k = k/s$. As k ranges from 0 to s, the values of t_k range from 0 to 1. The net result is that the domain D is replaced by the square $[0,1] \times [0,1]$ for plotting purposes.

Remark 20.3. The natural next step is to scale and translate the u values to the range $[0,1]$ so that the displayed graph lies entirely within the cube $[0,1]\times[0,1]\times[0,1]$. I will defer this to a (not so difficult) project in this chapter's *Projects* section. I have posed the problems *heat1* through *heat4* so that their u values lie more or less in the $[0,1]$ range to obviate a need for scaling. To view the solutions to arbitrary problems you will definitely need to scale.

[83] It's safe if you think of "whitespace" as the characters produced by the keyboard's *space-bar* or *tab* or *enter* keys. For the precise meaning of "whitespace" see subsection 9.3.2.

Listing 20.6: The implementation of the function `plot_curve()` in the file *heat-implicit.c*.

```
1  static void plot_curve(FILE *fp, double *u, int n, int steps, int k)
2  {
3      for (int j = 0; j < n+2; j++)
4          fprintf(fp, "%g %g %g\n",
5                  (double)k/steps, (double)j/(n+1), u[j]);
6  }
```

Listing 20.7: The implementation of the function `get_error()` in the file *heat-implicit.c*.

```
1  static double get_error(struct problem_spec *spec, double *u,
2          int n, double T)
3  {
4      double err = 0.0;
5      for (int j = 0; j < n+2; j++) {
6          double x = spec->a + (spec->b - spec->a)/(n+1)*j;
7          double diff = fabs(u[j] - spec->u_exact(x, T));
8          if (diff > err)
9              err = diff;
10     }
11     return err;
12 }
```

This has been a long narrative, but if you have followed the argument, you should have no difficulty in understanding how it results in the pleasingly brief implementation of `plot_curve()` shown in Listing 20.6.

20.9.5 ▪ The function `get_error()`

The function `get_error()` that appears on line 4 of Listing 20.3 is called only when an exact solution, $u_{\mathrm{ex}}(x,t)$, is supplied with the problem. It receives the vector $\boldsymbol{u}$ (see (20.24)) corresponding to the final time T and calculates and returns the error calculated according to (20.23). Examine the implementation in Listing 20.7, and be sure that you understand its details.

20.9.6 ▪ The function `heat_implicit()`

The function `heat_implicit()` that appears on line 7 of Listing 20.3 (page 272) contains the core of the implicit finite difference scheme. It advances an iterative loop from a time-slice t_k to the time-slice t_{k+1}, $k = 0, 1, 2, \dots$, by solving the system of equations (20.12) on page 257.

Conceptually, the values u_j^k form an $(n+2)\times(s+1)$ matrix, where $n+2$ and $s+1$ are the numbers of the grid points in the x and t directions. In our implementation we *do not* store that matrix u_j^k at all. Instead, we work with two vectors of length $n+2$ each, called the *current row* and the *next row*, that hold the rows k and $k+1$ of that matrix:

$$\begin{aligned}
&\text{current row:} \quad \boldsymbol{u} = \langle u_0^k, u_1^k, \dots, u_n^k, u_{n+1}^k \rangle,\\
&\text{next row:} \quad \boldsymbol{v} = \langle u_0^{k+1}, u_1^{k+1}, \dots, u_n^{k+1}, u_{n+1}^{k+1} \rangle.
\end{aligned}$$

Listing 20.8: The top half of the function `heat_implicit()` in the file *heat-implicit.c*.

```
1  static void heat_implicit(struct problem_spec *spec,
2          double T, int n, int steps, char *gv_filename)
3  {
4      FILE *fp;
5      double *u, *v, *d, *c;
6      double dx = (spec->b - spec->a)/(n+1);
7      double dt = T/steps;
8      double r = dt/(dx*dx);
9      if ((fp = fopen(gv_filename, "w")) == NULL) {
10         fprintf(stderr, "unable to open file '%s' for writing\n",
11                 gv_filename);
12         return;
13     }
14     fprintf(fp, "# geomview script written by the function %s()\n",
15             __func__);    // begin geomview script
16     fprintf(fp, "{ appearance { +edge }\n");
17     fprintf(fp, "MESH %d %d\n", n+2, steps+1);
18     printf("%g < x < %g,  0 < t < %g,  dx = %g, dt = %g,  "
19             "r = dt/dx^2 = %g\n",
20             spec->a, spec->b, T, dx, dt, r);
21     make_vector(u, n+2);
22     make_vector(v, n+2);
23     make_vector(d, n);
24     make_vector(c, n-1);
25     for (int j = 0; j < n+2; j++) {
26         double x = spec->a + (spec->b - spec->a)/(n+1)*j;
27         u[j] = spec->ic(x);
28     }
29     plot_curve(fp, u, n, steps, 0);
30     // continued in the next listing
```

Thus, given the current row, $\boldsymbol{u}$, that contains the solution at time t_k, we apply (20.12) to calculate the next row, $\boldsymbol{v}$, which will then contain the solution at time t_{k+1}. Then we *swap the two vectors*, whereupon the vector $\boldsymbol{u}$ will contain the solution at time t_{k+1} and the vector $\boldsymbol{v}$ will be available to receive the solution at time t_{k+2}. Thus, we march forward in time until we reach the desired final time destination. Since the vector $\boldsymbol{u}$ is overwritten at each time-step, we record the result by calling `plot_curve()` (which appends it to our *Geomview* script) before going on to the next step.

The implementation of the function `heat_implicit()` is shown in Listings 20.8 and 20.9. Let us examine the details.

Line 1. The parameter `spec` points to a **`struct`** `problem_spec` structure that contains the specification of the problem to be solved. The parameters `T`, `n`, `steps` are explained at the beginning of subsection 20.9.1 (page 272). The parameter `gv_filename` points to a string indicating a file name to which to write the computation's results in the form of a *Geomview* script.

Lines 9–17. We open a file to which to write our *Geomview* script. We print three lines to it according to the specification laid out in subsection 20.9.4. The first line, with the leading **#**, inserts a *comment* into the script for documentation purposes. The

Listing 20.9: The bottom half of the function `heat_implicit()` in the file *heat-implicit.c*.

```
31      // continued from the previous listing
32      for (int j = 0; j < n-1; j++)
33          c[j] = -r;
34      for (int k = 1; k ≤ steps; k++) {
35          double *tmp;
36          double t = T*k/steps;
37          v[0] = spec→bcL(t);
38          v[n+1] = spec→bcR(t);
39          u[1] += r*v[0];
40          u[n] += r*v[n+1];
41          for (int i = 0; i < n; i++)
42              d[i] = 1 + 2*r;
43          trisolve(n, c, d, c, u+1, v+1);
44          tmp = v;
45          v = u;
46          u = tmp;
47          plot_curve(fp, u, n, steps, k);
48      }
49      fprintf(fp, "}\n");  // end geomview script
50      fclose(fp);
51      printf("geomview script written to file %s\n", gv_filename);
52      if (spec→u_exact ≠ NULL) {
53          double err = get_error(spec, u, n, T);
54          printf("max error at time %g is %g\n", T, err);
55      }
56      free_vector(u);
57      free_vector(v);
58      free_vector(d);
59      free_vector(c);
60      putchar('\n');
61  }
```

comment contains the name of the current function, that is, `heat_implicit()`, which is captured by the `__func__` identifier. The `__func__` identifier[84] was introduced in C99. If you are limited to C89, then remove `__func__` and hard-code the function name instead. You may remove or change that comment, or add more such comment lines as you see fit; e.g., you may want to add your name and email address. To add the current date, see the documentation of `strftime()` in the standard library.

Line 18. We print a message to the *stdout* with the problem's basic data for the user's information. The transcript of the interactive session on page 266 shows how it manifests.

Lines 21–24. We allocate memory for the solver's working vectors. The vectors `u` and `v` are the *current row* and the *next row* vectors introduced earlier in this subsection. The vector `d` will hold the diagonal of the $n \times n$ tridiagonal matrix in (20.12)

---

[84]Since it may be difficult to discern in a typeset document, let me point out that there are *two leading and two trailing underscores* in `__func__`.

(page 257). The vector `c` will hold both the subdiagonal and the superdiagonal since the two are the same in this case.

Lines 25–29. We initialize the vector `u` with the problem's initial condition provided by the function `spec→ic()` and then call `plot_curve()` to insert that initial time-slice into our geomview script.

Continuing on to Listing 20.9:

Line 32. We initialize the super/subdiagonal vector `c` in accordance with system (20.12). As noted in Remark 20.2 (page 274), the vector `c` does not change during the solving process.

Lines 34–48. This is where the real action takes place. Given the *current row* `u`, it applies (20.12) to compute the *next row* `v` and repeats. Let me point out a few subtleties.

Lines 37–38. The first and last entries of the vector `v`, that is, `v[0]` and `v[n+1]`, fall on the domain's boundary; therefore their values are read from the supplied boundary conditions.

Lines 39–40. We adjust the vector `u` by adding the rightmost column of (20.12) to it.

Line 41. We initialize the diagonal vector `d` to $1+2r$, in accordance with (20.12). Unlike the vector `c`, which is not changed by `trisolve()`, the vector `d` does; therefore it needs to be refreshed upon each iterative pass.

Line 43. Having thus prepared the vectors `u`, `v`, `c`, and `d`, we pass them to `trisolve()` to solve the system (20.12). Although the state vectors `u` and `v` are of length $n+2$ each, the system (20.12) is only $n \times n$ since it *updates internal nodes only!* That's the reason for passing `u+1` and `v+1` to `trisolve()`; these skip the first elements of the two vectors. Their last elements are skipped as well since `trisolve()` accesses only the first `n` entries of the vectors it receives.

Lines 44–47. We swap the vectors `u` and `v` so that `u` now contains the problem's most up-to-date state and then call `plot_curve()` to plot that time-slice.

The rest of the code is self-explanatory, so I won't elaborate.

20.10 ▪ *Project Finite Differences in One Dimension*

Part 20.1. Implement, compile, and test *heat-implicit.c*.

Part 20.2. Copy *heat-implicit.c* to *heat-explicit.c*, and modify it to implement the explicit finite difference scheme of Section 20.2. If $r = \Delta t/(\Delta x)^2$ is greater than 0.5, print a message warning the user of the scheme's instability, but continue with the computation nevertheless.

Part 20.3. Copy *heat-implicit.c* to *heat-crank-nicolson.c*, and modify it to implement the Crank–Nicolson finite difference scheme of Section 20.4.

Part 20.4. Copy *heat-implicit.c* to *heat-seidman-sweep.c*, and modify it to implement the Seidman sweep finite difference scheme of Section 20.5.

You will find that unlike the previous iteration schemes that required a *current row* and a *next row* (the vectors `u` and `v`), you will be working with just one vector. The most obvious implementation of the Seidman scheme changes that vector *in place* as you sweep it from left to right and then from right to left.

Part 20.5. [optional] The function `trisolve()` computes the components of the vector `d[]`; see Listing 20.5, line 6. That vector, however, does not change during the program's execution. Therefore recomputing it upon every invocation of `trisolve()` is wasteful. See if you can change your program to compute `d[]` just once.

Part 20.6. [optional] Implement the suggestion of Remark 20.3.

Chapter 21

The porous medium equation

***Prerequisites:** Chapters 7, 8, 20*

21.1 ▪ Introduction

The *porous medium equation*

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u^m}{\partial x^2} \qquad (m > 1) \tag{21.1}$$

is a model of diffusion of a substance, typically gas, through a porous medium. The unknown $u(x,t)$ is the gas density at the point x at time t. The exponent m—which is not necessarily an integer—is a physical property of the diffusing material. Equation (21.1) arises in other contexts as well. Have a look at this chapter's appendix to see how it comes about as a model in population dynamics. You will find a quite readable treatise on the subject, along with an extensive bibliography, in Vázquez [77]. One thing you should know is that if the initial condition $u(x,0)$ is nonnegative, then the solution $u(x,t)$ is nonnegative for all $t > 0$, and therefore the exponentiation makes sense in that case. See Section 21.3 for a generalization to solutions of varying sign.

The goal in this chapter is to develop a finite difference scheme to solve initial/boundary value problems corresponding to (21.1). As we will see, the *Seidman sweep* scheme introduced in Section 20.5 leads to quite a simple implementation.

21.2 ▪ Barenblatt's solution

When $m = 1$, (21.1) reduces to the heat equation; cf. (20.1) on page 251. The condition $m > 1$, however, makes the porous medium equation a totally different beast compared to the heat equation. You may intuit a sign of trouble if you express (21.1) in the equivalent form $\frac{\partial u}{\partial t} = \frac{\partial}{\partial x}\big(m u^{m-1} \frac{\partial u}{\partial x}\big)$, which makes it clear that it is a nonlinear diffusion equation, as in $\frac{\partial u}{\partial t} = \frac{\partial}{\partial x}\big(\varkappa \frac{\partial u}{\partial x}\big)$, whose diffusion coefficient, $\varkappa = m u^{m-1}$, varies with the density. Since $m > 1$, $\varkappa$ tends to zero as density tends to zero. Thus, the diffusion process *degenerates* near zero densities. The porous medium equation is the archetype of the class of *degenerate parabolic equations*.

The bulk of the theory of parabolic equations concerns the nondegenerate case. Just about everything breaks down when degeneracy occurs. One of the striking facts about

the porous medium equation—a consequence of the degeneracy—is that its solutions exhibit a *finite speed of propagation* into vacuum (i.e., a zero density region). This is very much in contrast to the heat equation whose solutions spread out at infinite speed. The finite speed of propagation is evident in the class of self-similar solutions to (21.1) constructed by Barenblatt:

$$u(x,t) = \frac{1}{(t+\delta)^\beta}\left(\left[c - \gamma\left(\frac{x}{(t+\delta)^\beta}\right)^2\right]_+\right)^\alpha, \qquad -\infty < x < \infty, \quad t \geq 0, \tag{21.2}$$

where

$$\alpha = \frac{1}{m-1}, \quad \beta = \frac{1}{m+1}, \quad \gamma = \frac{m-1}{2m(m+1)},$$

and where $c \geq 0$ and $\delta \geq 0$ are arbitrary constants. The notation $[\,\cdot\,]_+$ means $\max(\cdot, 0)$. We see that at any time $t > 0$ the support[85] of the solution is

$$|x| < \sqrt{\frac{c}{\gamma}}\,(t+\delta)^\beta,$$

which propagates at a finite (but nonconstant) speed, as asserted.

In the special case of $m = 3$, Barenblatt's solution takes on a particularly simple form,

$$u(x,t) = \frac{1}{(t+\delta)^{1/4}}\sqrt{c - \frac{x^2}{12(t+\delta)^{1/2}}}, \tag{21.3}$$

or equivalently,

$$\frac{x^2}{12c(t+\delta)^{1/2}} + \frac{u^2}{c(t+\delta)^{-1/2}} = 1.$$

Therefore, the graph of u versus x (for any fixed t) is precisely the upper half of an ellipse with semimajor and semiminor axes lengths of $\sqrt{c(t+\delta)^{-1/2}}$ and $\sqrt{12c(t+\delta)^{1/2}}$. The ellipse flattens and spreads out as t increases. The area under the ellipse, which is proportional to the mass of the diffusing substance, remains constant at $\sqrt{3}\pi c$. Figure 21.1 shows snapshots of the graphs of $u(x,t)$ for several choices of t.

Barenblatt's solution is quite handy as a test case for our finite difference solver. We set up a problem with initial and boundary data derived from Barenblatt's solution, and then we expect that the solution produced by our solver will agree with Barenblatt's.

21.3 ▪ Generalizations

Since the exponent m in (21.1) is not necessarily an integer, the expression u^m is not well defined if u is negative. The extension

$$\frac{\partial u}{\partial t} = \frac{\partial^2(|u|^{m-1}u)}{\partial x^2} \tag{21.4}$$

of (21.1) admits negative u, is well-posed as a partial differential equation, and reduces to (21.1) when u is nonnegative. For this reason, most of the literature on the porous

[85] The *support* of a function is the closure of the set where it is nonzero.

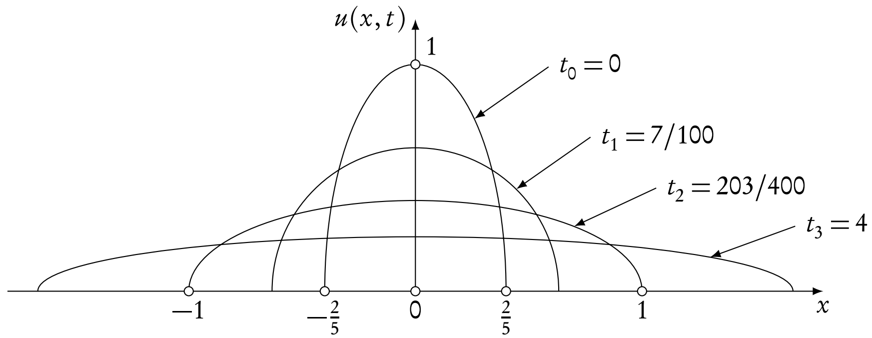


Figure 21.1: These flattening and spreading ellipses are snapshots of Barenblatt's solution, (21.3), of the porous medium equation with $m = 3$ at various times. The parameters $\delta = 1/75$ and $c = \sqrt{3}/15$ are chosen so that the solution at time $t_0 = 0$ has amplitude 1 and support $[-2/5, 2/5]$. I will leave it to you to verify that (a) at time $t_1 = 7/100$ the solution curve takes the form of a semicircle of radius $\sqrt{2/5}$, and (b) the solution front arrives at $x = \pm 1$ at time $t_2 = 203/400$. The solution curve at $t_3 = 4$ is also shown to give a better feel for how the solution spreads out in time.

medium equation addresses (21.4) rather than the special case (21.1). Yet a further generalization of (21.4) is the equation

$$\frac{\partial u}{\partial t} = \frac{\partial^2 \phi(u)}{\partial x^2}, \tag{21.5}$$

where $\phi : \boldsymbol{R} \to \boldsymbol{R}$ is some smooth monotonically increasing function with $\phi(0) = 0$ and $\phi'(0) = 0$. Much of the theory pertaining to the porous medium equation may be developed in the context of (21.5) with very minimal assumptions on ϕ. See, for instance, the articles [2, 9] and the book [77]. Clearly, (21.4) is the special case of (21.5) with $\phi(u) = |u|^{m-1}u$. I will state and explain this chapter's algorithm in the context of (21.5) for the sake of generality but will drop to the simple case of (21.1) for illustrations.

21.4 ▪ The finite difference scheme

The porous medium equation's counterpart of the initial/boundary value problem (20.2) is the following:

Find $u = u(x,t)$ so that

$$\frac{\partial u}{\partial t} = \frac{\partial^2 \phi(u)}{\partial x^2}, \qquad x \in (a,b),\ t > 0, \tag{21.6a}$$

$$u(x,0) = u_0(x), \qquad x \in (a,b), \tag{21.6b}$$

$$u(a,t) = u_L(t), \quad u(b,t) = u_R(t), \qquad t > 0. \tag{21.6c}$$

As in the case of Chapter 20's heat equation, the initial condition $u_0(x)$ and the left and right boundary conditions $u_L(t)$ and $u_R(t)$ serve to define a unique solution $u(x,t)$ in the semi-infinite strip $a \le x \le b$ and $t > 0$ in the x-t plane. Also as before, in a finite

difference approximation we replace the interval $a \le x \le b$ with a collection of $n+2$ equally spaced points $x_0 < x_1 < \cdots < x_{n+1}$, where $x_0 = a$ and $x_{n+1} = b$, and we let $\Delta x = (b-a)/(n+1)$. Similarly, we discretize the time into "time-slices" $t_0 < t_1 < t_2 \ldots$, where $t_0 = 0$, and the spacing between the slices is a prescribed Δt. Figure 20.2 on page 253 shows the resulting finite difference grid.

In the rest of this section I will use the notation and ideas introduced in Section 20.5 without further elaboration. The forward and reverse difference formulas (20.16) (on page 261) now take the form

$$v_j - u_j = -r\big(\phi(v_j) - \phi(v_{j-1})\big) - r\big(\phi(u_j) - \phi(u_{j+1})\big), \qquad j = 1, 2, \ldots, n, \tag{21.7a}$$

$$w_j - v_j = -r\big(\phi(v_j) - \phi(v_{j-1})\big) - r\big(\phi(w_j) - \phi(w_{j+1})\big), \qquad j = n, \ldots, 2, 1, \tag{21.7b}$$

where

$$r = \frac{\Delta t}{2(\Delta x)^2},$$

as in (20.15). (I have changed the notation from r' to r here since there is no chance of confusion within this chapter.) We rearrange the equations (21.7) into

$$v_j + r\phi(v_j) = r\phi(v_{j-1}) + u_j - r\phi(u_j) + r\phi(u_{j+1}), \qquad j = 1, 2, \ldots, n, \tag{21.8a}$$

$$w_j + r\phi(w_j) = r\phi(v_{j-1}) + v_j - r\phi(v_j) + r\phi(w_{j+1}), \qquad j = n, \ldots, 2, 1. \tag{21.8b}$$

During the forward sweep, all the terms on the right-hand side of (21.8a) are known. We solve the nonlinear equation $v_j + r\phi(v_j) =$ "known" to find v_j. Similarly, during the reverse sweep, all the terms on the right-hand side of (21.8b) are known. We solve the nonlinear equation $w_j + r\phi(w_j) =$ "known" to find w_j. Thus, in comparison with the heat equation of Chapter 20, the only extra effort is in solving a nonlinear equation of the form $\xi + r\phi(\xi) = c$ at each step. This may be accomplished through a Newton's iteration without much trouble by starting with an initial guess ξ_0. Since ϕ is expected to be a monotonically increasing function, any reasonable choice for ξ_0 will do. I suggest taking $\xi_0 = c$ and leave its implementation as an instructive project. In the rest of this chapter, however, I will focus on the special case of $\phi(u) = u^3$, which avoids Newton's iteration altogether. Here is why.

To solve the cubic equation $\xi + r\xi^3 = c$ for ξ, we multiply it through by $r^{1/2}$ to get $r^{1/2}\xi + r^{3/2}\xi^3 = r^{1/2}c$. Letting $\eta = r^{1/2}\xi$ and $k = r^{1/2}c$, we arrive at the cubic equation $\eta + \eta^3 = k$. You may verify that the unique real root of $\eta + \eta^3 = k$ has the explicit form

$$\eta = \frac{\gamma}{6} - \frac{2}{\gamma}, \quad \text{where} \quad \gamma = \left[108k + 12\sqrt{12 + 81k^2}\right]^{1/3}. \tag{21.9}$$

Thus, we evaluate η and then set $\xi = r^{-1/2}\eta = \eta/\sqrt{r}$.

21.5 ▪ The program

The rest of this chapter is devoted to details of the implementations of the Seidman sweep for solving the initial/boundary value problem (21.6) in the special case when $\phi(u) = u^3$. The case of a general ϕ is left as a project.

Our program relies on *xmalloc.[ch]* from Chapter 7 to allocate memory and the file *array.h* from Chapter 8 to construct vectors and matrices. Additionally, we will adapt the previous chapter's problem specification file, *problem-spec.c*, to the case in hand. The file *problem-spec.h* remains unchanged, so a symbolic link will do. Thus, following the suggestions in Chapters 2 and 6, the contents of the project's directory will look like this:

```
$ cd pme
$ ls -F
Makefile  pme-seidman-sweep.c  problem-spec.h@  xmalloc.h@
array.h   problem-spec.c       xmalloc.c@
```

Here is a transcript of a sample interactive session:

```
$ ./pme-seidman-sweep
Usage: ./pme-seidman-sweep T n s
   T : solve over 0 ≤ t ≤ T
   n : the number of grid points a=x[0], x[1], ... , x[n], x[n+1]=b
   s : the number of time slices 0=t[0], t[1], ... , t[s]=T

$ ./pme-seidman-sweep 1 20 20
problem pme1:
-1 < x < 1;  0 < t < 1,  dx = 0.0952381,
    dt = 0.05,  r = dt/(2*dx^2) = 2.75625
geomview script written to file pme1.gv
max error at time 1 is 0.013731

problem pme2:
-1 < x < 1;  0 < t < 1,  dx = 0.0952381,
    dt = 0.05,  r = dt/(2*dx^2) = 2.75625
geomview script written to file pme2.gv
```

The program solves two initial/boundary value problems for the porous medium equation, both with $\phi(u) = u^3$. These are the following:

Problem pme1: This produces a finite difference approximation to Barenblatt's solution (21.2) of the porous medium equation with $m = 3$, $c = \sqrt{3}/15$, and $\delta = 1/75$ on the bounded interval $-1 \leq x \leq 1$. The values of c and δ are chosen so that the initial condition (whose graph is the upper half of an ellipse) has amplitude 1 and is supported on the interval $[-2/5, -2/5]$. Figure 21.1 shows snapshots of the solution at a few selected times. Figure 21.2(left) shows the graph of the solution $u(x,t)$ as a surface in three dimensions.

Implementing the problem involves writing a function to evaluate Barenblatt's solution $u(x,t)$ in (21.2) and then extracting its initial and boundary data by evaluating $u(x,0)$ and $u(\pm 1, t)$, as explained in the "reverse-engineering" idea in (20.19) on page 264.

Problem pme2: This solves the initial/boundary value problem (21.6) on the interval $-1 \leq x \leq 1$ with the boundary data $u_L(t) = u_R(t) \equiv 0$ and the initial data

$$u_0(x) = \begin{cases} 1/2 & \text{if } 0 < x < 1/2, \\ -1/2 & \text{if } -1/2 < x < 0, \\ 0 & \text{otherwise.} \end{cases}$$

In particular, this tests the scheme's ability to handle initial data of variable sign. Figure 21.2(right) shows the graph of the solution $u(x,t)$ as a surface in three dimensions.

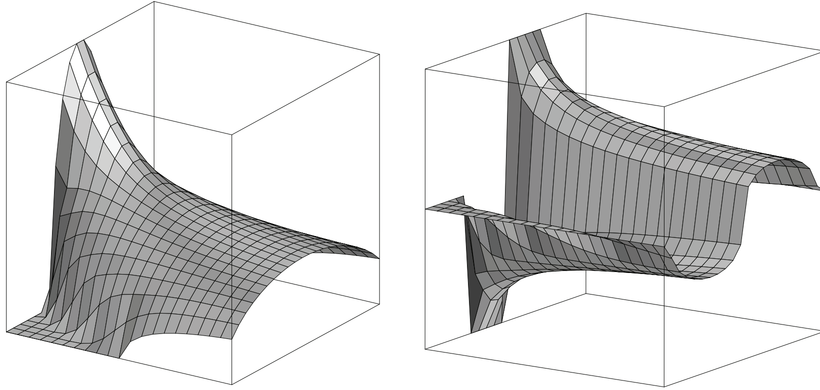


Figure 21.2: Graphs of the solutions $u(x,t)$ of problems pme1 (left) and pme2 (right) as surfaces in three dimensions computed by our finite difference solver and rendered in *Geomview*. Compare the graph of pme1 to the corresponding snapshots in Figure 21.1.

21.6 ▪ The files *problem-spec.[ch]*

The file *problem-spec.h* is identical to the file of the same name in the previous chapter, so a symbolic link will do. The file *problem-spec.c*, which is shown in an outline in Listing 21.1, contains the specifications of problems pme1 and pme2. The function barenblatt() on line 3 implements Barenblatt's solution (21.2), where m, c, and δ are given as parameters. The function pme1_exact(), which is given in its entirety beginning on line 7, produces a special case of Barenblatt's solution corresponding to $m=3$, $c=\sqrt{3}/15$, and $\delta = 1/75$. The functions pme1_ic(), pme1_bcL(), and pme1_bcR(), which yield the problem's initial and boundary data, simply evaluate pme1_exact() at $t=0$ and $x=\pm 1$. The function pme1() on line 15 is identical to the function heat1() given on lines 23–35 of Listing 20.2 on page 271 except for all occurrences of "heat1" being replaced by "pme1".

Similarly, functions pme2_ic(), pme2_ic(), pme2_bcL(), and pme2(), yield the problem specification of problem pme2 as described at the end of the previous section. You should have no problem in filling in the details. No exact solution is available for pme2.

21.7 ▪ The file *pme-seidman-sweep.c*

The file *pme-seidman-sweep.c* contains the implementation of the Seidman sweep scheme for solving the boundary value problem (21.6) assuming $\phi(u) = u^3$. The changes relative to the file *heat-seidman-sweep.c* of Chapter 20, which I assume that you wrote and tested earlier, are quite minimal. In effect, we need just one extra function, let's call it croot(), to solve the cubic equation $\eta + \eta^3 = k$ according to the formula (21.9). Here it is:

```
1  static double croot(double k)
2  {
3      double s = sqrt(12 + 81*k*k);
4      double gamma = pow(108*k + 12*s, 1.0/3);
5      return gamma/6 - 2/gamma;
6  }
```

Listing 21.1: An outline of the file *problem-spec.c*.

```
1  #include ...
2  // Barenblatt's solution
3  ▶static double barenblatt(double x, double t, double m,
4          double c, double delta) ...
5
6  // pme1: Barenblatt's solution with m = 3, and special choices of c and δ
7  static double pme1_exact(double x, double t)
8  {
9      double c = sqrt(3)/15, delta = 1.0/75;
10     return barenblatt(x, t, 3, c, delta);
11 }
12 ▶static double pme1_ic(double x) ...
13 ▶static double pme1_bcL(double t) ...
14 ▶static double pme1_bcR(double t) ...
15 ▶struct problem_spec *pme1(void) ...
16
17 // pme2: initial condition with variable sign
18 ▶static double pme2_ic(double x) ...
19 ▶static double pme2_bcL(double t) ...
20 ▶static double pme2_bcR(double t) ...
21 ▶struct problem_spec *pme2(void) ...
```

The **for**-loops that actually perform the forward or reverse sweeps are only slightly different from their Chapter 20 counterparts. Calculating v_j or w_j out of equations (20.17) (page 261) requires only a division by $1+r'$. Calculating v_j or w_j out of equations (21.8) (page 286) calls for solving a cubic equation. We have `croot()` to do that. Problem solved.

21.8 ▪ *Project Porous Medium*

Part 21.1. Copy the files *problem-spec.c* and *heat-seidman-sweep.c* from Chapter 20 to the current project's directory, and modify them as instructed above to produce a finite difference solver for the porous medium equation (21.6) with $\phi(u) = u^3$.

Part 21.2. [Optional] Modify your program to handle a general $\phi : \boldsymbol{R} \to \boldsymbol{R}$ assuming $\phi(0) = 0$ and ϕ is monotonically increasing. You will replace the function `croot()` with a solver (using Newton's iteration) for the equation $\xi + r\phi(\xi) = c$.

21.9 ▪ Appendix: The porous medium equation as a population dynamics model

It is not out of place to show how (21.6a) arises out of a simple population model. Although the treatment of this chapter has been limited to a one-dimensional space, the derivation of the model works more transparently in an n-dimensional setting, and that's what I will do. At the very end you may set $n = 1$ if you like.

Consider a certain hypothetical population that lives in the n-dimensional space $\boldsymbol{R}^n$. Assume that there exists a *population density function* $u(\boldsymbol{x}, t)$ so that the population in any

arbitrary region $\omega \subset \boldsymbol{R}^n$ at time t is given by $\int_\omega u(\boldsymbol{x},t)\,d\boldsymbol{x}$. Also assume that there exists a vector function $\boldsymbol{v}(\boldsymbol{x},t)$ that gives the velocity of the movement of the individuals at the point $\boldsymbol{x}$ at time t. For simplicity's sake, let's assume that the members of the population don't reproduce and don't die. Then the rate of increase of the population in ω equals exactly the inflow of the population through its boundary, that is,

$$\frac{d}{dt}\int_\omega u\,d\boldsymbol{x} = -\int_{\partial\omega} u\boldsymbol{v}\cdot\boldsymbol{n}\,da,$$

where $\boldsymbol{n}$ is the outward unit normal to the boundary $\partial\omega$ of ω.

We move the time derivative to under the integration sign. That's permissible since ω is independent of time. On the right-hand side, we may apply the *Divergence Theorem* (see, e.g., the section titled "Curl and Divergence" in Stewart [65]) from multivariable calculus to change the boundary integral to a volume integral. We get

$$\int_\omega \frac{\partial u}{\partial t}\,d\boldsymbol{x} = -\int_\omega \operatorname{div}(u\boldsymbol{v})\,da.$$

Since ω is arbitrary, we conclude that

$$\frac{\partial u}{\partial t} + \operatorname{div}(u\boldsymbol{v}) = 0.$$

What we have obtained is the *equation of conservation of mass*. That we derived it in the context of population dynamics is quite irrelevant. The density u and velocity $\boldsymbol{v}$ could have been those of the exhaust gases of a rocket, the air flowing around an airplane's wings, blood coursing through an animal's veins, or a vibrating metal plate. The equation of the conservation of mass links a material's velocity and density functions, assuming that the material is neither created nor destroyed.

Returning to the population model, assume that the species are averse to living in high density areas. More precisely, if the population density is not constant in a particular individual's neighborhood, the individual runs to a lower density area; that is, it moves in the opposite direction of the density gradient, ∇u, and the speed of the movement is a function of the density itself, let's say $\boldsymbol{v} = -\psi(u)\nabla u$. Substituting this in the equation of conservation of mass results in

$$\frac{\partial u}{\partial t} = \operatorname{div}\big[u\psi(u)\nabla u\big].$$

To connect this to the porous medium equation, introduce a function ϕ through $\phi(u) = \int_0^u \sigma\psi(\sigma)\,d\sigma$, that is, $\phi'(u) = u\psi(u)$ and $\phi(0) = 0$. Then the expression inside the square brackets becomes $\phi'(u)\nabla u$, that is, $\nabla\phi(u)$, and we conclude that

$$\frac{\partial u}{\partial t} = \operatorname{div}\nabla\phi(u) = \nabla^2\phi(u).$$

In the one-dimensional case this reduces to (21.6a).

Chapter 22
Gaussian quadrature

Prerequisites: *None*

22.1 ▪ Introduction

The Gaussian quadrature algorithm gives the optimal distribution of integration points and weights for integrating numerically a function of one variable on an interval. Specifically, an n-point quadrature rule on the interval $(-1,1)$ selects n points $\{x_j\}_{j=1}^n$, and weights $\{w_j\}_{j=1}^n$, so that for all reasonably nice functions $f(x)$ defined on $(-1,1)$ one has

$$\int_{-1}^{1} f(x)\,dx \approx \sum_{j=1}^{n} w_j f(x_j). \tag{22.1}$$

The interval $(-1,1)$ is the natural setting for developing the theory of the Gaussian quadrature, but that does not limit the algorithm's applicability; integrals on arbitrary intervals (a,b) may be reduced to integrals on $(-1,1)$ through a change of variables:

$$\int_a^b f(x)\,dx = \frac{b-a}{2}\int_{-1}^{1} f\left(\frac{b+a}{2} + \frac{b-a}{2}\xi\right) d\xi \tag{22.2a}$$

$$= \frac{b-a}{2}\int_{-1}^{1} f\left(\frac{a}{2}(1-\xi) + \frac{b}{2}(1+\xi)\right) d\xi. \tag{22.2b}$$

The two forms shown are obviously equivalent. Pick whichever you like.

To get a feel for what is meant by the "optimal distribution of integration points and weights" in this section's opening paragraph, let us consider the following (very) trivial thought experiment. Suppose we are interested in integrating *constant functions only*, that is, functions $f(x)$ such that $f(x) \equiv c$ for some c, independently of x. In that case $\int_{-1}^{1} f(x)\,dx = 2c = 2f(x^*)$. Any x^* will do since f is constant. Comparing to (22.1), this corresponds to a 1-point quadrature ($n=1$), with $w_1 = 2$ and $x_1 = x^*$.

There is no question that the formula $2f(x^*)$ gives *exact results* when f is constant. Also, there is no question that the formula $2f(x^*)$ won't work for nonconstant functions. Or will it? What if $f(x) = c_0 + c_1 x$ for some constants c_0 and c_1? Then we have

$$\int_{-1}^{1} f(x)\,dx = \int_{-1}^{1} (c_0 + c_1 x)\,dx = \left(c_0 x + \frac{1}{2}c_1 x^2\right)\Bigg|_{-1}^{1} = 2c_0 = 2f(0).$$

Surprise! The one-point quadrature rule with $w_1 = 2$ and $x_1 = 0$ gives *exact results* for all functions $f(x) = c_0 + c_1 x$.

In contrasting the two cases considered so far, we see that the one-point quadrature formula is able to integrate identically constant functions exactly. The choice of the quadrature point $x_1 = x^*$ is immaterial. However, the special choice of $x_1 = 0$ enables it to produce, *as a bonus*, the exact integrals of the nonconstant functions $f(x) = c_0 + c_1 x$ as well. This is the meaning of the "optimal distribution of integration points and weights" noted earlier. This observation generalizes in the following ways:

1. To any arbitrary choice of n distinct points $\{x_j\}_{j=1}^n$ in the interval $(-1,1)$, there corresponds weights $\{w_j\}_{j=1}^n$ so that

$$\int_{-1}^{1} p(x)\,dx = \sum_{j=1}^{n} w_j\, p(x_j) \quad \text{for all polynomials } p \text{ of degree } n-1. \tag{22.3}$$

2. There exists a *special choice* of n points $\{x_j\}_{j=1}^n$ in the interval $(-1,1)$, and weights $\{w_j\}_{j=1}^n$, so that

$$\int_{-1}^{1} p(x)\,dx = \sum_{j=1}^{n} w_j\, p(x_j) \quad \text{for all polynomials } p \text{ of degree } 2n-1. \tag{22.4}$$

We will justify these statements in sections 22.2 and 22.4. For now, let us note that the special choice of quadrature points yields a bonus whereby the quadrature's applicability extends from polynomials of degree $n-1$ to polynomials of degree $2n-1$. The earlier examples with $f(x) \equiv c$ and $f(x) = c_0 + c_1 x$ correspond to the case $n = 1$ of the statements above. There, the special choice of $x^* = 0$ extended the quadrature's applicability from polynomials of degree 0 to polynomials of degree 1.

Gaussian quadrature refers to the special choice of points and weights noted above. It is the goal of this chapter to explain some of the mathematics behind the Gaussian quadrature and then produce a C module that yields the lists of points and weights of a Gaussian quadrature of a desired degree on demand.

Remark 22.1. More often than not, we use Gaussian quadrature to integrate functions other than polynomials. In that case we obtain an approximation to the true value of the integral:

$$\int_{-1}^{1} f(x)\,dx \approx \sum_{j=1}^{n} w_j f(x_j).$$

The accuracy of the result depends on how close the function f is to a degree $2n-1$ polynomial. For detailed error estimates see [4, 28].

22.2 ▪ Lagrange interpolation

A generic polynomial $p(x) = c_0 + c_1 x + c_2 x^2 + \cdots + c_{n-1} x^{n-1}$ of degree $n-1$ in x is determined by its n coefficients $\{c_i\}_{i=0}^{n-1}$. It is within reason then to expect that a set of coefficients may be found so that $y_i = p(x_i)$, $i = 1, 2, \ldots, n$, where $\{(x_i, y_i)\}_{i=1}^n$ are given, and the x_i's are distinct.

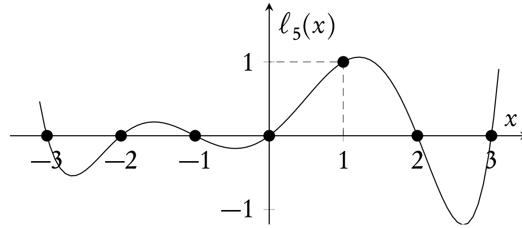


Figure 22.1: The graph of the sixth degree polynomial $\ell_5(x)$, with the property that $\ell_5(x) = 0$ for $x = -3, -2, -1, 0, 2, 3$, and $\ell_5(1) = 1$.

The brute force method of obtaining the coefficients $\{c_i\}_{i=0}^{n-1}$ is by solving the system of n linear equations $y_i = p(x_i)$ in the n unknowns c_i. The clever way of obtaining the coefficients is through *Lagrange interpolation*. To do the latter, consider the product

$$\pi_j(x) = \prod_{\substack{i=1\\ i\neq j}}^{n} (x - x_i) = (x - x_1)\cdots(x - x_{j-1})(x - x_{j+1})\cdots(x - x_n)$$

for $j = 1, 2, \ldots, n$, and note that $\pi_j(x)$ is a polynomial of degree $n-1$ in x, and $\pi_j(x_i) = 0$ for all i except for $i = j$, when it equals

$$\pi_j(x_j) = \prod_{\substack{i=1\\ i\neq j}}^{n} (x_j - x_i) = (x_j - x_1)\cdots(x_j - x_{j-1})(x_j - x_{j+1})\cdots(x_j - x_n).$$

It follows that the function

$$\ell_j(x) = \frac{\displaystyle\prod_{\substack{i=1\\ i\neq j}}^{n} (x - x_i)}{\displaystyle\prod_{\substack{i=1\\ i\neq j}}^{n} (x_j - x_i)} = \frac{(x - x_1)\cdots(x - x_{j-1})(x - x_{j+1})\cdots(x - x_n)}{(x_j - x_1)\cdots(x_j - x_{j-1})(x_j - x_{j+1})\cdots(x_j - x_n)}$$

is polynomial of degree $n-1$ in x, and

$$\ell_j(x_i) = \begin{cases} 0 & \text{if } i \neq j, \\ 1 & \text{if } i = j. \end{cases} \tag{22.5}$$

For instance, let the coordinates x_i be selected as the sequence of the seven values

$$[-3, -2, -1, 0, 1, 2, 3].$$

Since $x_5 = 1$, we have

$$\ell_5(x) = \frac{1}{48}(x+3)(x+2)(x+1)x(x-2)(x-3) \qquad (\textit{note:}\text{ no } (x-1) \text{ term!}).$$

Figure 22.1 shows the graph of the sixth degree polynomial $\ell_5(x)$. Note that $\ell_5(x)$ is zero when $x \in \{-3, -2, -1, 0, 2, 3\}$, and $\ell_5(1) = 1$, as expected.

Going back to the problem of determining the polynomial $p(x)$ of degree $n-1$ so that $y_i = p(x_i)$ for $i = 1, 2, \ldots, n$, it's clear that

$$p(x) = \sum_{j=1}^{n} \ell_j(x) y_j \tag{22.6}$$

fits the bill since it is of degree $n-1$, and $p(x_i) = y_i$ due to (22.5).

The polynomial $p(x)$ constructed above is unique. To see why, suppose that there is another $n-1$ degree polynomial, $q(x)$, such that $y_i = q(x_i)$ for $i = 1, 2, \ldots, n$. Let $r(x) = p(x) - q(x)$. Then we have $r(x_i) = 0$ for all $i = 1, 2, \ldots, n$. Thus, the $n-1$ degree polynomial r has n roots. By the *Fundamental Theorem of Algebra* that can happen only if $r(x)$ is identically zero. It follows that $p(x) \equiv q(x)$.

Equation (22.6) is called *Lagrange's interpolation formula*. It is important to note that the functions $\ell_j(x)$ *do not depend on the* y_j's. The y_j's appear only as *coefficients* in (22.6). Furthermore, since $y_i = p(x_i)$, it is convenient to express (22.6) in the form

$$p(x) = \sum_{j=1}^{n} \ell_j(x) p(x_j), \tag{22.7}$$

which removes the y_j's from the picture altogether. Then integrating over the interval $(-1, 1)$ we arrive at

$$\int_{-1}^{1} p(x)\,dx = \sum_{j=1}^{n} \left(\int_{-1}^{1} \ell_j(x)\,dx \right) p(x_j) \quad \text{for all polynomials } p \text{ of degree } n-1. \tag{22.8}$$

If we define the constants w_j according to

$$w_j = \int_{-1}^{1} \ell_j(x)\,dx, \quad j = 1, 2, \ldots, n,$$

then (22.8) exactly agrees with, and therefore proves, the statement (22.3).

22.3 ▪ Legendre polynomials

The *inner product* of two functions f and g over the interval $(-1, 1)$ is defined by

$$(f, g) = \int_{-1}^{1} f(x) g(x)\,dx.$$

The inner product extends the idea of the dot product of vectors to functions. The functions f and g are said to be *orthogonal* if $(f, g) = 0$. For instance, if $f(x) \equiv 1$, $g(x) = x$, and $h(x) = x^2$, then $(f, g) = \int_{-1}^{1} x\,dx = 0$, and therefore f and g are orthogonal. On the other hand, $(f, h) = \int_{-1}^{1} x^2\,dx = \frac{2}{3} \neq 0$, and therefore f and h are not orthogonal.

Starting with the sequence of monomials $1, x, x^2, x^3, \ldots$, we may apply the Gram–Schmidt orthogonalization procedure—see any textbook on linear algebra—to produce a sequence of orthogonal functions. The sequence of polynomials thus generated are called *Legendre polynomials*, the first few of which are

$$P_0(x) = 1, \quad P_1(x) = x, \quad P_2(x) = \frac{1}{2}(3x^2 - 1), \quad P_3(x) = \frac{1}{2}(5x^3 - 3x),$$
$$P_4(x) = \frac{1}{8}(35x^4 - 30x^2 + 3), \quad P_5(x) = \frac{1}{8}(63^5 - 70x^3 + 15x).$$

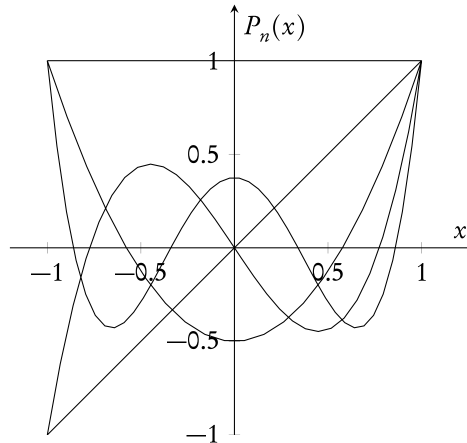


Figure 22.2: The graphs of the Legendre polynomials $P_n(x)$ for $n = 0, 1, 2, 3, 4$.

The orthogonality relationship is

$$(P_m, P_n) = \int_{-1}^{1} P_m(x)P_n(x)\,dx = \begin{cases} 0 & \text{if } m \neq n, \\ \frac{2}{2n+1} & \text{if } m = n. \end{cases}$$

There is a vast literature on the study of orthogonal polynomials, of which the Legendre polynomials are a very special case. If you are interested in learning more about these, Szegő [73] is a good starting point. You will find simplified presentations of some of the relevant concepts in the books by Atkinson [4] (particularly, equations (4.4.21), (4.4.26), and (5.3.28)) and Hildebrand [28] (particularly, equations (7.6.9), (8.5.5), and (8.5.8)). It turns out that for any n, the Legendre polynomial $P_n(x)$ has n distinct real roots, all of which lie in the interval $(-1, 1)$. Figure 22.2 shows the graphs of the first few Legendre polynomials.

Furthermore, it is possible to show that the Legendre polynomials and their derivatives satisfy the recursion equations

$$(n+1)P_{n+1}(x) = (2n+1)xP_n(x) - nP_{n-1}(x), \tag{22.9a}$$

$$(x^2 - 1)P_n'(x) = nxP_n(x) - nP_{n-1}(x). \tag{22.9b}$$

These, together with $P_0(x) \equiv 1$ and $P_1(x) = x$, provide a practical way of evaluating $P_n(x)$ and $P_n'(x)$ for all n and x.

22.4 ▪ The Gaussian quadrature formula

Consider a polynomial $p(x)$ of degree $2n - 1$ for some integer $n \geq 1$. By applying the Euclidean long division algorithm to the fraction $p(x)/P_n(x)$, we obtain

$$\frac{p(x)}{P_n(x)} = q(x) + \frac{r(x)}{P_n(x)},$$

where the quotient $q(x)$ and the remainder $r(x)$ are polynomials of degree $n - 1$ each. Rearranging the terms, we get

$$p(x) = P_n(x)q(x) + r(x). \tag{22.10}$$

Table 22.1: The coordinates x_j and weights w_j of the 4-point Gaussian quadrature.

j	x_j	w_j
1	−0.86113631159405257522	0.34785484513745385737
2	−0.33998104358485626480	0.65214515486254614263
3	0.33998104358485626480	0.65214515486254614263
4	0.86113631159405257522	0.34785484513745385737

Therefore

$$\int_{-1}^{1} p(x)\,dx = (P_n, q) + \int_{-1}^{1} r(x)\,dx.$$

The inner product (P_n, q) is zero since the nth degree polynomial $P_n(x)$ is orthogonal to all lower degree polynomials. Thus, the problem of integrating the $(2n-1)$-degree polynomial $p(x)$ reduces to the problem of integrating the $(n-1)$-degree polynomial $r(x)$. The latter problem, however, was thoroughly analyzed in Section 22.2 and led to the statement (22.8). Thus, we have

$$\int_{-1}^{1} p(x)\,dx = \sum_{j=1}^{n}\left(\int_{-1}^{1} \ell_j(x)\,dx\right) r(x_j). \tag{22.11}$$

Up to this point, the evaluation points $x_1, x_2, \dots, x_n$ have been arbitrary. Magic happens when we choose those to be the roots of the nth degree polynomial $P_n(x)$, that is, $P_n(x_j) = 0$, $j = 1, 2, \dots, n$. Then (22.10) indicates that $p(x_j) = r(x_j)$ for all j; therefore (22.11) takes the form

$$\int_{-1}^{1} p(x)\,dx = \sum_{j=1}^{n}\left(\int_{-1}^{1} \ell_j(x)\,dx\right) p(x_j) \quad \text{for all polynomials } p \text{ of degree } 2n-1,$$

which proves the statement (22.4).

The integral of $\ell_j(x)$, which yields the quadrature's weights w_j, may be evaluated in terms of the Legendre polynomials and their derivatives:

$$w_j = \int_{-1}^{1} \ell_j(x)\,dx = \frac{-2}{(n+1)P_n'(x_j)P_{n+1}(x_j)} = \frac{2}{(1-x_j^2)P_n'(x_j)^2}, \quad j = 1, 2, \dots, n.$$

See, e.g., Atkinson [4, page 276, equation (5.3.28)] and Hildebrand [28, page 391, equations (8.5.5) and (8.5.8)].

As a practical matter, the polynomial $P_n(x)$ and its derivative $P_n'(x)$ may be evaluated through the recursion formulas in (22.9). Then Newton's method may be applied to compute the roots x_j of $P_n(x)$. This supplies everything that is needed for evaluating the weights w_j from the formulas given above. See *Numerical Recipes in C* [53] for an implementation in C. Table 22.1 shows the coordinates x_j and weights w_j of the 4-point Gaussian quadrature for illustration.

22.5 ▪ The program

As noted above, it is possible to write a C program to compute the Gaussian quadrature coordinates x_j and the corresponding weights w_j on demand. However, since the computed data is static/unchanging, it makes better sense to compute them once for all and

Listing 22.1: An interactive session with the *gauss-quad-demo* program. The five problems that are solved here are described in this chapter's *Projects* section.

```
$ ./gauss-quad-demo
Usage:  ./gauss-quad-demo n
    n = number of quadrature points

$ ./gauss-quad-demo 2
integrating a cubic function over (-1, 1)
exact    = 2.6666666666666665
computed = 2.6666666666666665

integrating a cubic function over (2, 5)
exact    = 204.7500000000000000
computed = 204.7500000000000000

integrating a quadratic function over (-1,1) x (-1,1)
exact    = 6.66666666666666696
computed = 6.66666666666666607

integrating a quadratic function over (-1, 2) x (-1, 5)
exact    = 208.00000000000000000
computed = 208.00000000000000000

integrating sin(x) over (0, 3.14159)
exact    = 2.0000000000000000
computed = 1.9358195746511373
```

save them for future use. I have done that and have made the data file available at this book's website in the files *gauss-quad.[ch]*. Your job is to write a driver, let's call it *gauss-quad-demo.c*, to demonstrate the use of that data by performing a few integrations. This project's directory, therefore, will look like this:

```
$ cd gauss-quad
$ ls -F
Makefile gauss-quad-demo.c gauss-quad.c gauss-quad.h
```

My *gauss-quad-demo.c* contains five demonstrations. They are described in this chapter's *Projects* section. You are encouraged to modify these or add demonstrations of your own. Listing 22.1 shows a transcript of an interactive session.

22.6 ▪ The files *gauss-quad.[ch]*

I have computed the coordinates and weights of n-point Gaussian quadratures for $n = 1, \dots, 15$ and placed them in the file *gauss-quad.c*, which you may download from this book's website. I did the calculations in Maple with 40 digits of accuracy and then printed the results with 20 digits of accuracy, as in `"%.20f"`. The associated header file *gauss-quad.h* declares a structure

```
struct Gauss_qdat {
    double p;         // point
    double w;         // weight
};
```

that holds a point/weight pair. An n-point quadrature table may be stored in an array of length n of such structures. I do, however, store the table in an array of length $n+1$. The $(n+1)$st extra entry is a dummy `{0.0, -1.0}` point/weight pair which serves as the table's terminal sentinel. In scanning the table, we recognize the sentinel by its negative weight value since the normal Gauss quadrature weights are never negative. The inclusion of sentinel is by no means essential. I have introduced it merely for computational convenience. You will see its use in the code fragment at the end of this section.

In view of this, Table 22.1's 4-point quadrature data is then encoded as the C structure:

```
static struct Gauss_qdat quaddata4[] = { /* n = 4 */
    {   -0.86113631159405257522,    0.34785484513745385737},
    {   -0.33998104358485626480,    0.65214515486254614263},
    {    0.33998104358485626480,    0.65214515486254614263},
    {    0.86113631159405257522,    0.34785484513745385737},
    {                       0.0,                      -1.0},
};
```

This is one of the 15 quadrature tables defined in *gauss-quad.c*; they provide quadrature data for n-point Gaussian quadratures for any $n = 1, 2, \ldots, 15$. The tables are named `quaddata1[]`, `quaddata2[]`, ..., `quaddata15[]`.

To provide further organization to the data tables, I have defined an array, called `QuadTables[]`, each of whose entries points to one of the data tables:

```
static struct Gauss_qdat *QuadTables[] = {
    NULL,           // no 0-point quadrature
    quaddata1,
    quaddata2,
    ...
    quaddata15
};
```

Thus, `Gauss_qdat[n]` evaluates to a pointer to the `n`-point quadrature table.

You may observe that the quadrature tables and the `QuadTables[]` array are all specified **`static`**; therefore they are not directly accessible outside *gauss-quad.c*. Access to the data is provided through the front-end function `gauss_qdat()`, which is defined in *gauss-quad.c* as

```
struct Gauss_qdat *gauss_qdat(int *n)
{
    int ntables = (sizeof QuadTables / sizeof QuadTables[0]) - 1;
    if (*n < 1)
        *n = 1;
    else if (*n > ntables)
        *n = ntables;
    return QuadTables[*n];
}
```

The function begins with calculating `ntables`, which is the number of available quadrature tables. (See Remark 4.2 on page 22 for an explanation of that calculation.) We know that there are 15 tables, but that number is not hard-coded in the function since it may change in the future.

The parameter `n` points to an integer n which the caller sets as the number of desired quadrature points. If the requested number falls outside the `1 ... ntables` range, it is brought into the range first (silently), and then a pointer to an appropriate table is returned to the caller. Since the value stored in `*n` may have to be changed during this call,

the function receives a *pointer* to an integer rather than an integer in its argument. The caller may examine the value of `*n` after the function returns if that value is of concern.

The following code fragment illustrates the use of our quadrature tables in integrating a function f over the interval $[-1,1]$:

```
struct Gauss_qdat *gqdat;
int n = 10;                      // want a 10-point quadrature
gqdat = gauss_qdat(&n);          // get the quadrature table
sum = 0.0;
for (int i = 0; i < n; i++)
    sum += gqdat[i].w * f(gqdat[i].p);
printf("integral = %g\n", sum);
```

Alternatively, we may use the table's sentinel for the stopping criterion:

```
struct Gauss_qdat *gqdat;
int n = 10;                      // want a 10-point quadrature
gqdat = gauss_qdat(&n);          // get the quadrature table
sum = 0.0;
while (gqdat->w != -1) {         // watch for the sentinel
    sum += gqdat->w * f(gqdat->p);
    gqdat++;
}
printf("integral = %g\n", sum);
```

I (slightly) prefer the second version's index-free code, but that's a matter of taste. Do whichever way that makes better sense to you.

22.7 ▪ Project Gaussian Quadrature

Part 22.1. Write a program *gauss-quad-demo.c* which reads the number n of quadrature points from the command-line and applies Gaussian quadrature to integrate the cubic function $f(x) = 1 + x + x^2 + x^3$ over the interval $[-1,1]$. The exact answer is $8/3$. Can you tell, before running your program, the minimal number of quadrature points that would produce the exact answer within the machine's floating point accuracy?

Part 22.2. Extend *gauss-quad-demo.c* to compute, additionally, the integral of the previous part's f over the interval $[2,5]$. You will use one or the other of the change of variables formulas in (22.2). The exact answer is $819/4 = 204.75$.

Part 22.3. According to Fubini's Theorem, the double integral over the square $[-1,1] \times [-1,1]$ may be expressed as nested single integrals:

$$\int_{-1}^{1}\int_{-1}^{1} f(x,y)\,dx\,dy = \int_{-1}^{1}\left(\int_{-1}^{1} f(x,y)\,dx\right)dy.$$

If we approximate each of the single integrals on the right-hand side through a Gaussian quadrature, we get

$$\int_{-1}^{1}\int_{-1}^{1} f(x,y)\,dx\,dy \approx \sum_{i=1}^{m} w_i\Big(\sum_{j=1}^{n} w_j f(x_j,y_i)\Big) = \sum_{i=1}^{m}\sum_{j=1}^{n} w_i w_j f(x_j,y_i).$$

Extend *gauss-quad-demo.c* to apply the Gaussian quadrature with $m = n$ to approximate the integral of the quadratic function $g(x,y) = 1 + x + y + x^2 + xy + y^2$ over the square $[-1,1] \times [-1,1]$. The exact answer is $20/3$.

Part 22.4. Extend *gauss-quad-demo.c* to integrate the previous part's g over the rectangle $[-1,2] \times [1,5]$. The exact answer is 208.

Part 22.5. Extend *gauss-quad-demo.c* to compute $\int_0^\pi \sin x \, dx$. The exact answer is 2. Since the integrand is not a polynomial, a Gaussian quadrature can only produce an approximate answer. Examine the answer's accuracy as the number of quadrature points increases. *Note:* Don't hard-code the numerical value of π in your program. Instead, use `4*atan(1)`.

Chapter 23

Triangulation with the *Triangle* library

Prerequisites: *Chapters 7, 8*

23.1 ▪ Introduction

Triangulating a planar polygonal domain means dividing the domain into a union of triangles so that

1. every triangle has a nonempty interior;
2. no two triangles have common interior parts; and
3. any side of any triangle is either a part of the domain's boundary or is the side of another triangle.

Figure 23.1 shows sample triangulations. We see that a polygonal domain may have a number of holes. Thus, a "polygonal domain" means a planar region bounded by one or more nonintersecting polygonal boundaries.

A triangulated domain is also called a *triangular mesh*. A "mesh" is a more general concept than triangulation; one may have a mesh consisting of quadrilaterals or hexagons, for example. In this book we will work with triangular meshes only; therefore when I say a "mesh", I mean a triangular mesh.

Triangulated domains are used frequently in the finite element method (FEM) for solving partial differential equations (PDEs). (More on that in Chapter 25.) Stated very loosely, the solution to the PDE is approximated by a low degree polynomial on each triangle. The FEM sees to it that the polynomial patches connect continuously across adjacent edges. The coefficients of the polynomials are adjusted to obtain the best possible solution.

Producing good triangulations is a complicated matter and an active area of research. A good triangulation—the technical term is a "quality triangulation"—produces triangles whose angles are neither too small nor too large. We don't want needle-like triangles; the accuracy of the finite element approximation depends on that.

Triangle is a C library written by *Jonathan Shewchuk* [60, 61] for producing quality triangulations of planar polygons, possibly with holes. It is available in source form from `<https://www.cs.cmu.edu/~quake/triangle.html>`. We will use *Triangle* for meshing in this and subsequent chapters. See Section 23.7 on how to install *Triangle* in your working environment.

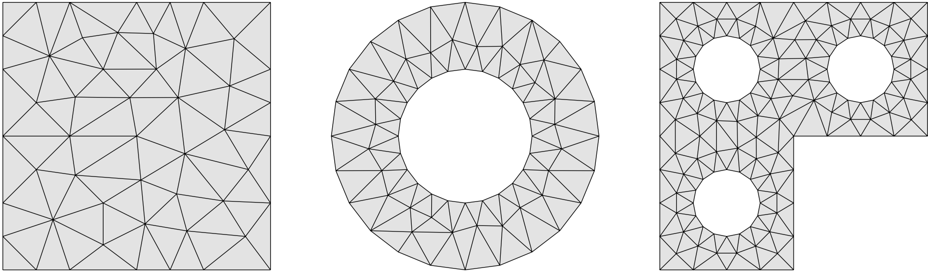


Figure 23.1: Sample triangulations. The inner and outer boundaries of the annulus are 24-sided polygons. In the figure on the right, each hole is a 16-sided polygon.

In this chapter I will introduce two distinct data structures which may be thought of as *Triangle*'s "data before meshing" and "data after meshing". The "before" data structure, introduced in Section 23.3, specifies a domain's geometry. For instance, a square is specified by its four vertices and the four edges that connect them. The "after" data structure, introduced in Section 23.5, captures the triangulation's data, that is, the list of the mesh's vertices and the triangle edges that connect them.

Both data structures carry additional information, some of which does not pertain to the meshing per se but is essential to FEM applications. To explain, let us look at an elementary boundary value problem for a PDE:

$$\begin{cases} \nabla \cdot \big(\eta(x,y)\nabla u(x,y)\big) + f(x,y) = 0 & \text{in } \Omega, \\ u = g & \text{on } \Gamma_D, \\ \eta \dfrac{\partial u}{\partial n} = h & \text{on } \Gamma_N. \end{cases} \tag{23.1}$$

Here $\nabla = \langle \partial/\partial x, \partial/\partial y\rangle$ is the gradient operator, Ω is a planar domain, Γ_D and Γ_N are two complementary parts of its boundary, and $\partial u/\partial n$ is the derivative of u in the outward normal direction at the boundary. Solving the boundary value problem amounts to finding a function $u(x,y)$ that satisfies the PDE in Ω (first line), the *Dirichlet* boundary condition on Γ_D (second line), and the *Neumann* boundary condition on Γ_N (third line). The functions η, f, g, and h are given. The "before" data structure specifies not only the geometry of Ω, but also the boundary parts Γ_D and Γ_N, because the triangle edges that fall on the boundary are to inherit the boundary data from Γ_D or Γ_N. The functions η, f, g, and h are not relevant to the domain's triangulation per se; therefore they will play no part in this chapter. Nevertheless, we will have placeholders in our data structures for them so that we won't have to redo these data structures in the later chapters when we take on the task of solving the boundary value problem (23.1).

The goal of this chapter is to introduce the "before" and "after" data structures for the boundary value problem (23.1) and enough details about the *Triangle* library to enable you to mesh arbitrary polygonal domains with its help. *Solving* the boundary value problem will have to wait for a future chapter.

23.2 ▪ The program

This chapter's programs rely on the *xmalloc* module (Chapter 7) for managing memory and the *array.h* header file (Chapter 8) for creating vectors and matrices. Additionally,

you will need the files *triangle.[ch]*, *mesh-to-eps.[ch]*, and *mesh.c* from the book's website. I will describe their purposes later. Then, following the suggestions in Chapters 2 and 6, the initial state of this project's directory will look like

```
$ cd meshing
$ ls -F
Makefile        mesh-to-eps.h    problem-spec.h  xmalloc.h@
array.h@        mesh.c           triangle.o@
mesh-demo.c     mesh.h           triangle.h@
mesh-to-eps.c   problem-spec.c   xmalloc.c@
```

You will write *mesh.h*, *problem-spec.[ch]*, and *mesh-demo.c* as you read through this chapter. The symbolic links *triangle.[ho]* are explained in Section 23.7.

Let us say the program compiles into the executable file *mesh-demo*. Here is a transcript of a sample session:

```
$ ./mesh-demo
Usage: ./mesh-demo a
    a = maximal triangle area

$ ./mesh-demo 0.02
domain is a square
vertices = 52, edges = 130, elems = 79
PostScript output written to file square.eps

domain is a triangle with hole
vertices = 90, edges = 220, elems = 130
PostScript output written to file triangle-with-hole.eps

domain is an annulus (really a 24-gon) of radii 0.325 and 0.65
vertices = 74, edges = 174, elems = 100
PostScript output written to file annulus.eps

domain is an L-shape with three holes (actually 16-gons
vertices = 141, edges = 349, elems = 206
PostScript output written to file three-holes.eps
```

In the first attempt, the program is invoked without an argument. It writes a message saying that it expects an argument a which prescribes the *maximal triangle area* which sets an upper bound on the areas of the mesh's triangles and thus controls the fineness of the mesh. In the second attempt the program is invoked with an argument of `0.02`. We see in the transcript that it meshes four distinct domains and prints the numbers of vertices, edges, and elements in each case. An "element" in this context means a triangle; that's a harbinger of future uses of this utility for FEM analysis.

Additional information in the transcript indicates that the program writes an image of each mesh into a *PostScript* file. These may be viewed with any *PostScript* viewer.

23.3 ▪ The file *problem-spec.h*

It was noted in Section 23.1 that the program deals with separate "before meshing" and "after meshing" data structures. In this and the next section I will describe the former. In Section 23.5 I will describe the latter.

The "before meshing" data structure is called "**struct** `problem_spec`" because it carries a *problem specification*. It is declared in the file *problem-spec.h*, which is shown in its entirety in Listing 23.1. Let us see what we have there:

Listing 23.1: The file *problem-spec.h*.

```
#ifndef H_PROBLEM_SPEC_H
#define H_PROBLEM_SPEC_H
#define FEM_BC_DIRICHLET    2
#define FEM_BC_NEUMANN      3
struct problem_spec_point {
    int point_no;
    double x;
    double y;
    int bc;
};
struct problem_spec_segment {
    int segment_no;
    int point_no_1;
    int point_no_2;
    int bc;
};
struct problem_spec_hole {
    double x;
    double y;
};
struct problem_spec {
    struct problem_spec_point *points;
    struct problem_spec_segment *segments;
    struct problem_spec_hole *holes;
    int npoints;
    int nsegments;
    int nholes;
    double (*f)(double x, double y);
    double (*g)(double x, double y);
    double (*h)(double x, double y);
    double (*eta)(double x, double y);
    double (*u_exact)(double x, double y);
};
#endif /* H_PROBLEM_SPEC_H */
```

Lines 3 and 4. The two preprocessor macros declared here are used later to label the domain's boundary parts Γ_D and Γ_N corresponding to the Dirichlet and Neumann boundary conditions in the boundary value problem (23.1). There is nothing special about the numbers 2 and 3 associated with these macros; they could have been any two distinct integers other than `0` and 1. The exceptions are due to the special meanings that *Triangle* attaches to boundary markers with values `0` and 1.

Line 5. The domain's boundary consists of vertices and the edges that connect them. Within *Triangle* these are called *points* and *segments*.[86] The structure declared on line 5 is designed to hold a point specification consisting of an *index* (`point_no`), which serves as the point's label, the point's x and y coordinates, and its boundary

[86] Segments are more general than what I am presenting here. *Triangle* allows for *internal segments*. These may be used, for instance, to divide a domain into *regions*. One may produce a mesh whose fineness varies from region to region. We will not have a use for that feature in this book.

condition type `bc`. Indices are consecutive integers beginning with zero, assigned (by the user) to the domain's points in no particular order. The boundary condition type is one of `FEM_BC_DIRICHLET` or `FEM_BC_NEUMANN`.

Line 11. The structure declared here is designed to hold a segment specification consisting of an *index* (`segment_no`), which serves as the segment's label, the indices of the points at the segment's ends, and the segment's boundary condition `bc`. Indices are consecutive integers beginning with zero, assigned (by the user) to the domain's segments in no particular order. The boundary condition type is one of `FEM_BC_DIRICHLET` or `FEM_BC_NEUMANN`.

Line 17. The structure declared here is designed to hold information about the domain's holes (see Figure 23.1 for examples of domains with holes). A hole is identified through the coordinates (x, y) of an arbitrary point within it. Here is how this works. *Triangle* triangulates the entire domain, including the holes, but it's careful not to make any triangles that straddle a hole's boundary. Then, beginning at the hole's (x, y) identifier, it "pops" every triangle it encounters until it reaches the hole's boundary.

Line 21. The structure declared here is a wrapper around the previous three structures. It also includes (a) the members `npoints`, `nsegments`, and `nholes`, which hold the counts of the domain's points, segments, and holes, respectively; and (b) the members `f`, `g`, `h`, `eta`, `u_exact`, which are pointers to functions of type $\boldsymbol{R}^2 \to \boldsymbol{R}$. The program will set these to point to the functions f, g, h, η, u that enter the formulation of the boundary value problem (23.1). The final member, `u_exact`, requires some explanation. In general, the solution u of (23.1) is not known ahead of the time; otherwise there is no point in writing a program to compute it. For testing purposes, however, it is a good idea to solve a few problems whose exact solutions *are known* ahead of the time. This provides an opportunity to test the code for accuracy or find bugs. The `u_exact` member of the `problem_spec` structure points to the exact solution if one is available.

The members `f`, `g`, `h`, `c`, `u_exact` do not affect a domain's triangulation. They are placeholders for objects which will come into play in later chapters when we implement the FEM for solving the boundary value problem (23.1).

23.4 ▪ The file *problem-spec.c*

The file *problem-spec.c* may contain any number of problem specification. My current *problem-spec.c* contains four problems. I will describe two in detail, and I will let you write the other two. Listing 23.2 gives an outline of *problem-spec.c*.

23.4.1 ▪ Triangle with a hole

The domain shown in Figure 23.2 is an isosceles triangle with a rectangular hole. In the diagram on the left I have labeled the seven vertices, n_0 through n_6, and the seven edges, e_0 through e_6. The enumeration order is immaterial, but the indices should begin at zero and increase in steps of 1. I have superimposed a rectangular grid over the diagram on the left to help you read the vertex coordinates. The grid's lower-left corner is at $(-1, 0)$. The diagonally opposite corner is at $(1, 2)$. The diagrams labeled (b) and (c) show the grid

Listing 23.2: An outline of the file *problem-spec.c*.

```
1  ▶ #include ...
2  ▶ struct problem_spec *triangle_with_hole(void) ...  // see Listing 23.3
3  ▶ struct problem_spec *annulus(int n) ...            // see Listing 23.4
4  ▶ void free_annulus(struct problem_spec *spec) ...   // see Listing 23.5
5  ▶ struct problem_spec *square(void) ...
6  ▶ struct problem_spec *three_holes(int n) ...
7  ▶ void free_three_holes(struct problem_spec *spec) ...
```

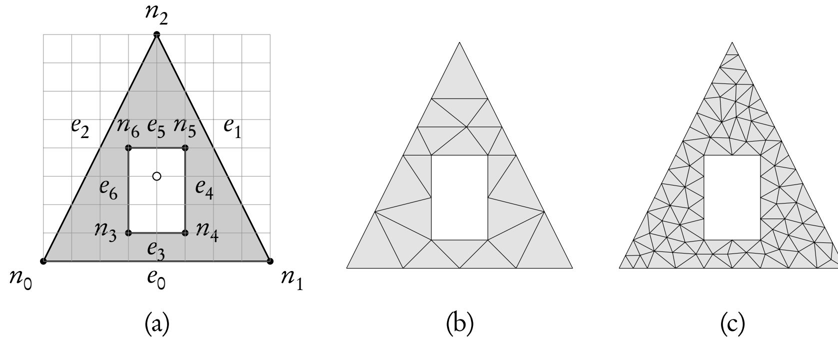


Figure 23.2: (a) Triangle with a hole before meshing; (b) after meshing with the maximum area parameter set to 0.2; and (c) after meshing with maximum area parameter set to 0.02. In (a) I have superimposed a rectangular grid to help you read the vertex coordinates. The grid's lower-left corner is at $(-1,0)$. The diagonally opposite corner is at $(1,2)$. The hollow circle at $(0,0.75)$ marks an arbitrary point within the hole. *Triangle* relies on this to recognize the inside of a hole.

generated by the program corresponding to setting the maximum area of triangles to 0.2 and 0.02, respectively.

The function `triangle_with_hole()` that appears on line 2 of Listing 23.2 returns a pointer to a `problem_spec` structure corresponding to the domain in Figure 23.2(a). Its implementation is shown in Listing 23.3. Let's examine the details.

Line 3. We define and initialize the array `points[]` of the following type: "**`struct`** `problem_spec_point`". That structure, described in Section 23.3, holds a point's index, its x and y coordinates, and its boundary condition type. Since our domain has seven vertices/points, the array `points[]` is of length 7. Lines 5 through 12 define those points.

The **`static`** specifier in the declaration of `points[]` means that the array persists throughout the program's lifetime (that is, from when the program is invoked until when in exits). Thus, a pointer to the structure *remains meaningful after the function* `triangle_with_hole()` *returns*. In the absence of the **`static`** specifier, the array would have come into existence upon entering the function and would have vanished upon exiting; therefore the address returned by the function would have been meaningless.

Listing 23.3: The implementation of the function `triangle_with_hole()` in the file *problem-spec.c*.

```
1  struct problem_spec *triangle_with_hole(void)
2  {
3      static struct problem_spec_point points[] =  { // the points
4          // triangle's vertices
5          { 0,    -1.0,   0.0,    FEM_BC_DIRICHLET },
6          { 1,    1.0,    0.0,    FEM_BC_DIRICHLET },
7          { 2,    0.0,    2.0,    FEM_BC_DIRICHLET },
8          // hole's vertices
9          { 3,    -0.25,  0.25,   FEM_BC_DIRICHLET },
10         { 4,    +0.25,  0.25,   FEM_BC_DIRICHLET },
11         { 5,    +0.25,  1.0 ,   FEM_BC_DIRICHLET },
12         { 6,    -0.25,  1.0 ,   FEM_BC_DIRICHLET },
13     };
14
15     static struct problem_spec_segment segments[] = { // the segments
16         // triangle's segments
17         { 0,    0, 1,   FEM_BC_DIRICHLET },
18         { 1,    1, 2,   FEM_BC_DIRICHLET },
19         { 2,    2, 0,   FEM_BC_DIRICHLET },
20         // hole's segments
21         { 3,    3, 4,   FEM_BC_DIRICHLET },
22         { 4,    4, 5,   FEM_BC_DIRICHLET },
23         { 5,    5, 6,   FEM_BC_DIRICHLET },
24         { 6,    6, 3,   FEM_BC_DIRICHLET },
25     };
26
27     static struct problem_spec_hole holes[] = { // the hole's identifier
28         { 0.0, 0.75 }
29     };
30
31     static struct problem_spec spec = {   // C99-style initialization!
32         .points     =  points,
33         .segments   = segments,
34         .holes      = holes,
35         .npoints    = sizeof points / sizeof points[0],
36         .nsegments  = sizeof segments / sizeof segments[0],
37         .nholes     = sizeof holes / sizeof holes[0],
38         .f          = NULL,
39         .g          = NULL,
40         .h          = NULL,
41         .eta        = NULL,
42         .u_exact    = NULL,
43     };
44
45     printf("domain is a triangle with hole\n");
46     return &spec;
47 }
```

Line 15. We define and initialize the array `segments[]` of the following type: "**struct** problem_spec_segment". That structure, described in Section 23.3, holds a segment's index, the indices of the points at its ends, and its boundary condition type. Since our domain has seven edges/segments, the array `segments[]` is of length 7. Lines 17 through 24 define those segments. The previous paragraph's comments on the **static** specifier apply here as well.

Line 27. We define and initialize the array `holes[]` of the following type: "**struct** problem_spec_hole". That structure, described in Section 23.3, holds the x and y coordinates of an arbitrary point inside a hole. Since our domain has one hole, `holes[]` is an array of length 1. I have picked $(0, 0.75)$ as this hole's identifier point; see Figure 23.2(a). The previous paragraph's comments on the **static** specifier apply here as well. See the next paragraph regarding domains with no holes.

Line 31. We define and initialize `spec`, an object of type "**struct** problem_spec", which was described in Section 23.3. Its first three members are the addresses of the `points[]`, `segments[]`, and `holes[]` arrays defined earlier. Its next three members are the lengths of those arrays. I could have entered 7, 7, and 1 for those lengths, but I didn't. Instead, I let the program itself calculate the lengths following the idea noted in Remark 4.2 on page 22. This makes the program more robust because if I choose to insert or delete a point in the `points[]` array, I won't have to update the entry here. The rest of the structure's members are pointers to the various functions that enter the definition of the boundary value problem (23.1). These are irrelevant to meshing, so I am setting them to `NULL` for now.

Remark 23.1. If the domain has no holes, then set `.holes` to `NULL` and `.nholes` to `0`.

Line 45. The informational message printed here is by no means essential; however, it helps make the function a little more user-friendly. Inspect the transcript of the interactive session in Section 23.2 to see where the message "domain is a triangle with hole" appears.

23.4.2 ▪ An annulus

Figure 23.3 shows an annular domain bounded by two octagons. In the diagram on the left, I have labeled the domain's 16 vertices, n_0 through n_{15}, and the 16 edges, e_0 through e_{15}. The enumeration order is immaterial, but the indices should begin at zero and increase in steps of 1. The distances of the inner points from the center are 0.325. The distances of the outer points from the center are twice that. There is no particular reason for these choices other than the desire to make the area of the annulus to be approximately 1 so that we may compare its triangulation in a meaningful way to that of a 1×1 square. On the right, I have shown the mesh produced by invoking the program with the argument 0.02.

The function `annulus()` that appears on line 3 of Listing 23.2 on page 306 returns a pointer to a `problem_spec` structure corresponding to the domain in Figure 23.3(a). Its implementation is shown in Listing 23.4. Unlike `triangle_with_hole()`, which takes no arguments, `annulus()` takes one argument, n, and produces n-gons for the annulus's inner and outer boundaries. The domain in Figure 23.3 was produced by calling `annulus(8)`.

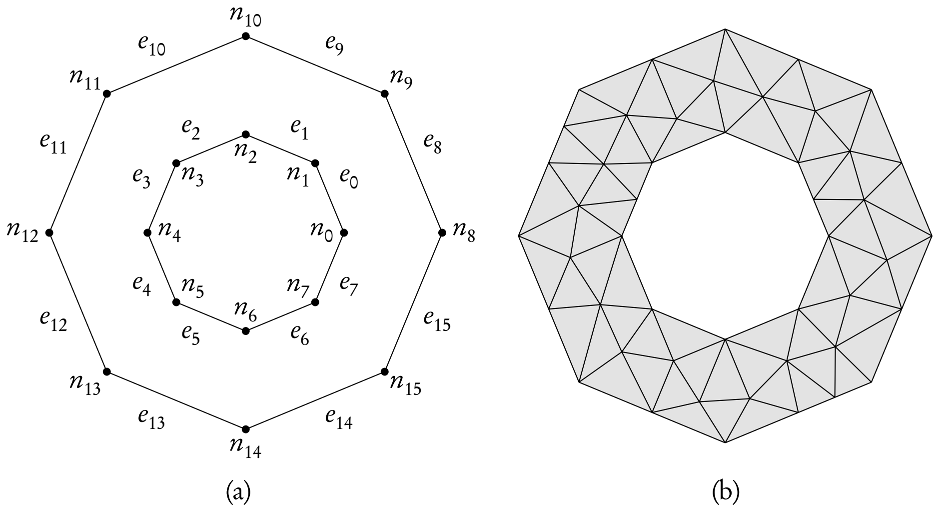


Figure 23.3: (a) An octagonal annulus before meshing; (b) mesh produced by `annulus(8)` with maximum area parameter set to 0.02. In (a) I have shown my labeling of the points and segments.

There is a fundamental difference between the functions `triangle_with_hole()` and `annulus()`. In the former, the geometry is fixed once for all. I took advantage of that when I *combined the declaration and initialization* of the `points[]` array (see line 3 in Listing 23.3 on page 307). I treated the `segments[]` and `holes[]` arrays (Listing 23.3, lines 15 and 27) in the same way. In contrast, the geometry of the annulus is unknown ahead of the time since it depends on the argument n. That prevents us from combining the definitions and initializations of the arrays. Instead, we call `xmalloc()` to allocate arrays of appropriate lengths according to n and then populate their entries one at a time in **`for`**-loops. Additionally, we supply a function, `free_annulus()`, that the user will call to free the allocated memory when it's no longer needed. Let us examine the details.

Line 3. We need the number π to calculate the vertex coordinates. We have $\tan \pi/4 = 1$, and therefore $\pi = 4\tan^{-1} 1$. Here we apply the standard library's `atan()` function to evaluate the inverse tangent.

Line 4. We define the annulus's inner and outer radii a and b. You may change these numbers if you wish. Better yet, you may want to modify `annulus()` so that it receives a and b as arguments, as in `annulus(a,b,n)`.

The choice of $a = 0.325$ and $b = 2a$ results in an annulus of area of $\pi(b^2 - a^2) \approx 0.995$. I wanted to get an area close to 1 to compare its triangulation to that of a 1×1 square. There is no deep reason for that choice.

Line 5. Here we declare "`spec`" as a pointer to "**`struct`** `problem_spec`" and at the same time allocate memory for the structure.

Lines 6–8. We allocate vectors/arrays to hold the domain's geometry. The first two vectors are of length $2n$ each because the domain has $2n$ vertices and $2n$ edges. The third vector is of length 1 because there is only one hole.

Listing 23.4: The implementation of the function `annulus()` in the file *problem-spec.c*.

```
struct problem_spec *annulus(int n)
{
    double Pi = 4*atan(1);
    double a = 0.325, b = 2*a;
    struct problem_spec *spec = xmalloc(sizeof *spec);
    make_vector(spec->points, 2*n);
    make_vector(spec->segments, 2*n);
    make_vector(spec->holes, 1);

    for (int i = 0; i < n; i++) { // define the points
        double t = 2*i*Pi/n;
        spec->points[i].point_no = i;
        spec->points[i].x = a*cos(t);
        spec->points[i].y = a*sin(t);
        spec->points[i].bc = FEM_BC_DIRICHLET;
        spec->points[i+n].point_no = i+n;
        spec->points[i+n].x = b*cos(t);
        spec->points[i+n].y = b*sin(t);
        spec->points[i+n].bc = FEM_BC_DIRICHLET;
    }

    for (int i = 0; i < n; i++) { // define the segments
        spec->segments[i].segment_no = i;
        spec->segments[i].point_no_1  = i;
        spec->segments[i].point_no_2  = i+1;
        spec->segments[i].bc = FEM_BC_DIRICHLET;
        spec->segments[i+n].segment_no = i+n;
        spec->segments[i+n].point_no_1  = i+n;
        spec->segments[i+n].point_no_2  = i+n+1;
        spec->segments[i+n].bc = FEM_BC_DIRICHLET;
    }
    spec->segments[n-1].point_no_2 -= n;
    spec->segments[2*n-1].point_no_2 -= n;

    spec->holes[0].x = spec->holes[0].y = 0.0;   // the hole's identifier

    spec->npoints = 2*n;
    spec->nsegments = 2*n;
    spec->nholes = 1;
    spec->f = spec->g = spec->h = spec->eta = spec->u_exact = NULL;
    printf("domain is an annulus (really a %d-gon) "
            "of radii %g and %g\n", n, a, b);
    return spec;
}
```

Listing 23.5: The function `free_annulus()` in the file *problem-spec.c*.

```
1  void free_annulus(struct problem_spec *spec)
2  {
3      if (spec ≠ NULL) {
4          free_vector(spec→points);
5          free_vector(spec→segments);
6          free_vector(spec→holes);
7          free(spec);
8      }
9  }
```

Line 10. In a **`for`**-loop we define the vertex coordinates through

$$\begin{cases} x_i = a\cos\dfrac{2\pi i}{n}, \\ y_i = a\sin\dfrac{2\pi i}{n}, \end{cases} \qquad \begin{cases} x_{n+i} = b\cos\dfrac{2\pi i}{n}, \\ y_{n+i} = b\sin\dfrac{2\pi i}{n}, \end{cases} \qquad i = 0, 1, \dots, n-1.$$

Thus, the inner vertices are labeled 0 through $n-1$, and the outer vertices are labeled n through $2n-1$. Figure 23.3(a) illustrates the $n = 8$ case.

Line 22. The **`for`**-loop defines the segment i as connecting vertex i to vertex $i+1$ for $i = 0, 1, \dots, 2n-1$. This is not quite correct, however, since the segment $n-1$ should connect the vertex $n-1$ to vertex 0, as the polygon closes onto itself. The same problem exists with the segment $2n-1$. We fix these in the two lines that immediately follow the **`for`**-loop.

Line 35. Here we define the identifier point of the domain's hole. I have taken the obvious choice $(0, 0)$, but any other point within the hole, e.g. $(a/2, 0)$, would have been just as good.

Line 37. We are done with defining the domain's components. We assign the `spec` structure's remaining members, print an informative message, and return `spec` to the caller.

This completes the details of the function `annulus()`. In Listing 23.5 we provide a companion function, `free_annulus()`, to enable the user to free the memory allocated within `annulus()`. The function enters on line 4 of Listing 23.2 on page 306.

23.5 ▪ The files *mesh.h* and *mesh.c*

At this point we have a framework for specifying a domain's geometry, such as that shown in Figure 23.3(a). In this section we introduce a companion framework to address a domain's triangulation, such as that seen in Figure 23.3(b).

The data structures for holding the triangulation's data are declared in the file *mesh.h* shown in Listing 23.6. Let us examine it.

Line 4. A **`struct`** `node` holds data for a *node* of the mesh. For the purposes of this book, you may think of a node as a triangle's vertex. In more general FEM applications, nodes may include points other than vertices.

Listing 23.6: The file *mesh.h*.

```
#ifndef H_MESH_H
#define H_MESH_H
#include "problem-spec.h"
struct node {
    int nodeno;
    double x;
    double y;
    double z;
    int bc;
};
struct edge {
    int edgeno;
    struct node *n[2];        // pointers to the edge's endpoints
    int bc;
};
struct elem {
    int elemno;
    struct node *n[3];        // pointers to the element's nodes
    struct edge *e[3];        // pointers to the element's edges
    double ex[3], ey[3];      // x and y components of the edge vectors
    double area;              // element's area
};
struct mesh {
    struct node *nodes;       // the array of node structures
    struct edge *edges;       // the array of edge structures
    struct elem *elems;       // the array of elem structures
    int nnodes;
    int nedges;
    int nelems;
};
struct mesh *make_mesh(struct problem_spec *spec, double a);
void free_mesh(struct mesh *mesh);
#endif /* H_MESH_H */
```

We see that the node structure has members to hold a node's index, a boundary condition type (`bc`), and the node's (x, y) coordinates. There is no use for the z member as far as meshing goes. Later, when we solve PDEs, z will hold the value of the solution at the point (x, y).

Line 11. A **`struct`** `edge` holds data for an *edge* of the mesh. An edge is the side of a triangle. It connects two nodes. The edge structure has members to hold an edge number, a boundary condition type (`bc`), and an array, `n[2]`, that holds two pointers to the node structures corresponding to the edge's endpoints.

Line 16. A **`struct`** `elem` holds data for an *element* of the mesh. I use "element" as a synonym for "triangle" in this context. The element structure has a member `elemno` to hold an element's number; an array `n[3]` to hold pointers to the node structures of the element's vertices; and an array `e[3]` to hold pointers to the edge structures of the element's edges. Furthermore, the arrays `ex[3]` and `ey[3]` hold the x and y components of the element's *edge vectors*, which are depicted in Figure A.1 on page 376. The `area` member holds the element's area.

Remark 23.2. The data stored in the `ex[]`, `ey[]`, and `area` members are redundant insofar as they can be computed on the fly from the element's node coordinates which are available through the element structure's `n[]` member. We will find out, however, that the aforementioned data are needed in multiple places in finite element applications. To avoid the repeated recomputation of the data, we compute them once for all and store them in the `ex[]`, `ey[]`, and `area` members for later use.

Line 23. A **`struct`** `mesh` is a wrapper around the previous three structures. It also includes additional data consisting of the integers `nnodes`, `nedges`, and `nelems` that hold the counts of the domain's nodes, edges, and elements, respectively.

Line 31. The function `make_mesh()` presents an extremely simple interface to meshing a domain. It receives a pointer to a "**`struct`** `problem_spec`" that contains the details of the domain to be meshed. It calls *Triangle* to mesh the domain and returns the result as a pointer to a "**`struct`** `mesh`". (It allocates memory for that structure as necessary.) The parameter `a` is taken from the commend-line; see Section 23.2. It specifies an upper bound on the areas of the mesh's triangles. A smaller `a` will result in a finer mesh.

Line 32. The function `free_mesh()` frees all memory allocated by `make_mesh()`. The user is responsible for calling `free_mesh()` when those memory resources are no longer needed.

The function `make_mesh()` hides a great deal of work behind it. First, it translates the domain description from a **`struct`** `problem_spec` to a format that *Triangle* understands. Next, it calls *Triangle* to mesh the domain. And finally, it translates *Triangle*'s output into a **`struct`** `mesh`. Thus, the meshing activity takes place entirely behind the scenes. The user of `make_mesh()` is isolated completely from the internal workings of *Triangle*. This is a *good thing*—it enables one to replace *Triangle* with an alternative triangulator without affecting the rest of the program.

The file *mesh.c* contains the implementation of `make_mesh()`. Although it is not very complex, I will refrain from explaining its details here because there is little pedagogical value in it. Instead, I have placed the file in the book's website for you to download, compile, and link with your programs. Feel free to examine it if you are curious. It's not very long. You will need to know something about *Triangle*'s own data structures in order to understand how it works.

23.6 ▪ The file *mesh-demo.c*

The file *mesh-demo.c* provides a demo of our meshing module. It calls `make_mesh()` on the domains defined in *problem-spec.c* and writes the resulting meshes as EPS images which may be viewed with any *PostScript* viewer. The fineness of the mesh is determined by the user-supplied maximal triangle area on the command-line. Listing 23.7 provides an outline of *mesh-demo.c*. Let us look at it, beginning with `main()`:

Lines 16–21. These supply the prototypes of the functions `square()`, `annulus()`, etc., which are defined in *problem-spec.c*. I could have included those prototypes in the file *problem-spec.h*, but I decided against it since adding new functions in *problem-spec.c* would necessitate changing *problem-spec.h*, which would not be pretty. I have

Listing 23.7: An outline of the file *mesh-demo.c*.

```
1  ▶#include ...
2  static void do_demo(struct problem_spec *spec,
3                          double a, char *eps_filename)
4  {
5      struct mesh *mesh = make_mesh(spec, a);
6      printf("vertices = %d, edges = %d, elems = %d\n",
7              mesh→nnodes, mesh→nedges, mesh→nelems);
8      mesh_to_eps(mesh, eps_filename);
9      free_mesh(mesh);
10 }
11
12 ▶static void show_usage(char *progname) ...
13
14 int main(int argc, char **argv)
15 {
16     struct problem_spec *square(void);
17     struct problem_spec *triangle_with_hole(void);
18     struct problem_spec *annulus(int n);
19     void free_annulus(struct problem_spec *spec);
20     struct problem_spec *three_holes(int n);
21     void free_three_holes(struct problem_spec *spec);
22     struct problem_spec *spec;
23     char *endptr;
24     double a;
25
26     ▶check argc, call show_usage() and exit if other than 2
27     a = strtod(argv[1], &endptr);
28     ▶check endptr, call show_usage() and exit if bad
29
30     do_demo(triangle_with_hole(), a, "triangle-with-hole.eps");
31     putchar('\n');
32
33     spec = annulus(24);
34     do_demo(spec, a, "annulus.eps");
35     free_annulus(spec);
36     putchar('\n');
37
38     ▶also do square()
39     ▶also do three_holes()
40     return EXIT_SUCCESS;
41 }
```

chosen to include the prototypes in `main()` since that's the only place where they are needed.

Lines 26–28. The program is to be invoked with exactly one argument, `a`, which specifies the mesh's maximal triangle area. Therefore, we expect `argc` to be 2. We apply the standard library's `strtod()` function (see Chapter 5) to extract `a`. Call `show_usage()` and exit if the argument is malformed or if it represents a non-positive number.

Line 30. To mesh Figure 23.2's triangular domain, we call `triangle_with_hole()` (see section 23.4.1) to retrieve the domain's `problem_spec` structure and pass the result immediately to the function `do_demo()` (to be described later) for meshing. We also pass to `do_demo()` the value of `a`, which it will need, and the string "*triangle-with-hole.eps*", which is the file name to which it will write the mesh's EPS image. The file name is arbitrary; any legitimate file name will do.

Lines 33–36. We repeat the process with `annulus()`, which was defined in section 23.4.2. Here I am invoking it as `annulus(24)`, but you should experiment with varying the argument to get a feel for what it does. Try, for instance, `annulus(8)`. That will get you the octagonal annulus of Figure 23.3.

Unlike the previous case, I have separated the invocations of `annulus()` and `do_demo()` since I want to capture the pointer `spec` returned by `annulus()`, which I will need to pass to `free_annulus()` on line 35 to free the structure's memory. This was not necessary in the previous case since the data structures there were not allocated dynamically.

Lines 38 and 39. The contents of these additional demos are the subject of Section 23.8.

Line 12. The function `show_usage()` is called when the user invokes the program improperly. It prints a brief usage instruction to the `stderr`; see the transcript of the sample session shown in Section 23.2.

Line 2. The function `do_demo()` defined here is quite straightforward. It receives a pointer to a `problem_spec` structure, which it passes to `make_mesh()` to create a mesh. Then it prints the counts of the nodes, edges, and elements thus obtained for the user's information. Next, it passes the mesh structure to `mesh_to_eps()`, which writes an image of the mesh in the *PostScript* format to the file named in its argument. Finally it calls `free_mesh()` to free the memory resources associated with the mesh structure and returns.

The function `mesh_to_eps()` is declared and implemented in the files *mesh-to-eps.[ch]*, which you downloaded from the book's website. Implementing the function `mesh_to_eps()` is not a part of this project because that requires some familiarity with the *PostScript* language, which I don't want to make a prerequisite for this book. Just compile the downloaded files along with the rest of your files and use `mesh_to_eps()` as a "black box".

23.7 ▪ Installing *Triangle*

To compile this chapter's programs, you will need a copy of *Triangle*, which you may get from its home website noted in Section 23.1. It won't work out of the box, however, since it requires setting a few preprocessor options. I suggest that you use the modified version that I have provided in the book's website. My modifications amount to setting the preprocessor options that make it into a C module rather than a stand-alone program. The triangulation code itself is not touched.

The file *triangle.h* is the module's interface. It includes extensive comments on the module's use. The file *triangle.c* (which is over 16,000 lines long!) contains the module's implementation (and built-in documentation for the stand-alone version). The code implements a few kludges in the interest of minimizing memory usage; therefore you will

get warnings when you compile it with gcc's `-Wall -pedantic` flags. Nevertheless, the compiled program seems to work on the machines I have tested.

You may drop the files *triangle.h* and *triangle.c* into your project's directory and treat them like any **.c* and **.h* file of your own. Although that will work, it's not a good idea. First, mixing *triangle.h* and *triangle.c* with your own files is ugly. Second, *triangle.h* and *triangle.c* are needed in several other projects. It makes no sense to keep multiple copies of them. Third, *triangle.c* is quite large and it takes several seconds to compile even on a fast machine. It will slow you down if you have to compile it often.

To address these issues, I suggest that you put *triangle.h* and *triangle.c* in a directory of their own, let's say *../triangle/*, relative to your current project's directory. Go to that directory, and compile *Triangle* once for all:

```
$ cd ../triangle/
$ cc -Wall -pedantic -std=c89 -O2 triangle.c -c
```

This will produce the object file *triangle.o*. There will be several warnings due to the kludges noted above. Ignore the warnings.

Now go to your project's directory and establish symbolic links to *Triangle*:

```
$ cd ../meshing/
$ ln -s ../triangle/triangle.[ho] .
```

(Don't overlook the space and the period at the end of that command.) This makes the files *triangle.h* and *triangle.o* available to your project. Compile and link with the rest of your project's files as usual.

Remark 23.3. If you know how to make libraries on your operating system, you may build a true library out of *triangle.o* and install it, along with *triangle.h*, in standard places where the compiler looks for libraries and header files. Then you may link your program with *Triangle* simply by giving a `-ltriangle` flag to the compiler at the linking stage. There are too many variations and platform dependencies on making and installing libraries; therefore I cannot afford to go into that subject in any depth. If you don't know how, ask someone versed in the use of your platform to show you.

23.8 ▪ *Project Triangulate*

Listing 23.2 on page 306 gives an outline of the file *problem-spec.c*. Some of its parts have been explained in the preceding sections. Complete the rest of that file according to the instructions below.

Part 23.1. The function `square()` in Listing 23.2 defines a domain in the form of a simple square $(0,1) \times (0,1)$.

1. Add the definition of `square()` to *problem-spec.c*.
2. Insert the necessary code in *mesh-demo.c* (see line 38 of Listing 23.7) that calls `do_demo()` to mesh the domain. A sample result is shown in Figure 23.1 on page 302.

Part 23.2. The function `three_holes()` in Listing 23.2 defines an L-shaped domain with three holes removed, as shown in Figure 23.1 on page 302. Each hole is an n-gon, where n is received as an argument.

1. Add the definition of `three_holes()` to *problem-spec.c*.
2. Insert the necessary code in *mesh-demo.c* (see line 39 of Listing 23.7) that calls `do_demo()` to mesh the domain.

Suggestion: Set the domain's reentrant vertex at $(0,0)$ and the other vertices at $(s,0)$, (s,s), $(-s,s)$, $(-s,-s)$, and $(0,-s)$. Let $r = s/4$ be the hole radius. You may pick any number you wish for s. I used $s = 0.64$ because that makes the domain's area approximately equal to 1.

Chapter 24

Integration on triangles

Prerequisites: *Chapters 7, 8, 23, and Appendix A*

24.1 ▪ Introduction

The Gaussian quadrature algorithm of Chapter 22 gives the optimal distribution of quadrature points and weights for integrating numerically a function of one variable on an interval. It identifies the points and weights as simple functions of the roots of certain Legendre polynomials.

Unfortunately no such definitive procedure exists for integrating a function of two variables over a triangle. True, one may regard the triangle as the image of a square, and the square as the product of two intervals, and thus extend the Gaussian quadrature to a triangle. However, the distribution of the quadrature points obtained this way is far from optimal, as evidenced by the illustration in Figure 24.1; the quadrature points crowd into one vertex, requiring many essentially redundant function evaluations there. An efficient quadrature scheme will distribute the quadrature points more evenly on the triangle. The purpose of this chapter is to introduce one such quadrature scheme that has been developed by Taylor, Wingate, and Bos [75, 74].

An n-point quadrature rule on a triangle T selects n points $\{(x_i, y_i)\}_{i=1}^n$, and weights $\{w_i\}_{i=1}^n$, so that for all reasonably nice functions $f(x,y)$ defined on T one has

$$\int_T f(x,y)\,dx\,dy \approx \sum_{i=1}^n w_i f(x_i, y_i).$$

Cartesian coordinates, however, do not mesh well with triangles. Barycentric coordinates (see Appendix A) work much better, and that's what we will use most of time when dealing with triangles.

A point inside a triangle may be expressed in Cartesian coordinates (x,y) or barycentric coordinates $(\lambda_1, \lambda_2, \lambda_3)$, where $0 \le \lambda_1, \lambda_2, \lambda_3 \le 1$ and $\lambda_1 + \lambda_2 + \lambda_3 = 1$. There is a one-to-one correspondence between the Cartesian and barycentric coordinates. Therefore, to a function $f(x,y)$ expressed in Cartesian coordinates there corresponds a function $\tilde{f}(\lambda_1, \lambda_2, \lambda_3)$ in barycentric coordinates so that $f(x,y) = \tilde{f}(\lambda_1, \lambda_2, \lambda_3)$, where (x,y) and $(\lambda_1, \lambda_2, \lambda_3)$ refer to the same point. The n-point quadrature formula shown above

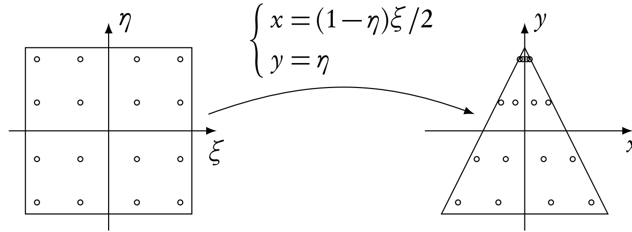


Figure 24.1: The diagram shows a mapping of the square $(-1,1)\times(-1,1)$ in the (ξ,η) plane to a triangle in the (x,y) plane. The (4×4)-point Gaussian quadrature on the square maps to a certain quadrature scheme on the triangle. The distribution of the triangle's quadrature points, however, is far from optimal—they crowd into one vertex.

may be expressed in the equivalent form

$$\int_T f(x,y)\,dx\,dy \approx \sum_{i=1}^{n} w_i \tilde{f}\big(\lambda_1^{(i)},\lambda_2^{(i)},\lambda_3^{(i)}\big),$$

where $\big(\lambda_1^{(i)},\lambda_2^{(i)},\lambda_3^{(i)}\big)$ are the barycentric coordinates of the quadrature points.

The benefit of formulating the quadrature in terms of the barycentric coordinates is that once we determine a good set of quadrature points on *one* triangle—any triangle will do—the result extends to *all* triangles. This is because any triangle may be mapped onto any another through a linear transformation, and barycentric coordinates are invariant under linear mappings (subsection A.1.2).

The quadrature weights, however, are proportional to the triangle's area since the Jacobian determinant of a linear transformation is constant and measures the ratio of the areas. The usual practice is to tabulate the weights corresponding to a *standard reference triangle*, let's call it T_{std}, and then adjust the quadrature formula for arbitrary triangles by multiplying by the area ratios. This leads to the quadrature formula

$$\int_T f(x,y)\,dx\,dy \approx \frac{|T|}{|T_{\text{std}}|}\sum_{i=1}^{n} w_i \tilde{f}\big(\lambda_1^{(i)},\lambda_2^{(i)},\lambda_3^{(i)}\big), \tag{24.1}$$

where $|T|$ and $|T_{\text{std}}|$ are the areas of the triangle T and the standard reference triangle T_{std}.

Generally a larger n gives a better approximation. The optimal placement of the quadrature points and their weights depends on n.[87] The coarsest approximation is obtained with $n=1$, which calls for sampling the function *in just one point* to estimate the value of the integral. How good is the formula (24.1) if we pick the sampling point, that is, $\big(\lambda_1^{(1)},\lambda_2^{(1)},\lambda_3^{(1)}\big)$, arbitrarily in T and take $w_1=|T_{\text{std}}|$? Certainly not very good but not entirely useless either. For instance, it will give *exact results for constant functions.* (Do you see that?) We can do even better. If we sample the function at the triangle's *centroid*, then it is not hard to see that (24.1) with $n=1$ will give exact results not only for constants but also for *all first degree polynomials!* In that sense, the choice of the quadrature point $(\lambda_1,\lambda_2,\lambda_3)=(1/3,1/3,1/3)$, which picks up the centroid, and the weight $w=|T_{\text{std}}|$ is optimal for a one-point quadrature.

[87] In that regard I should have written $\big(\lambda_1^{(n,i)},\lambda_2^{(n,i)},\lambda_3^{(n,i)}\big)$ and $w_i^{(n)}$ to make the dependence on n explicit, but I didn't want to complicate the notation.

This brings up a question: What is the smallest number of quadrature points, n, for which the formula (24.1) gives exact answers for *all quadratic polynomials?* There is no reason to stop at quadratics. We may ask the question of polynomials of any degree. For instance, what is the smallest number of quadrature points, n, for which the formula (24.1) gives exact answers for *all polynomials of degree 14?*

A general answer to such questions is not known. There are various algorithms for computing n and the corresponding points and weights, but there is no assurance that they actually are optimal. For instance, according to Table 2 of [74], three different algorithms in [13], [78], and [74] for exact quadrature of polynomials of degree 14 call for 42, 46, and 45 points, respectively. It is not known if 42 points is the fewest possible.

The goal of this chapter is to learn how to integrate on triangles with the help of the quadrature data developed by Taylor, Wingate, and Bos [75, 74]. Unlike most other chapters of this book, this chapter's study *does not* lead to a self-contained module. The reason is that in applications such as finite element models, integrations take place at the innermost level of multiply nested loops. It is significantly more efficient to insert a customized integration code directly in the heart of the nested loops rather than pass data back and forth to a general external module to evaluate the integrals. Consequently, we will use this chapter's *ideas and techniques*, rather than its C code, in future chapters.

24.2 ▪ The Taylor, Wingate, and Bos (TWB) quadrature

The article [74] describes the algorithm used by Taylor, Wingate, and Bos to determine quadrature data (i.e., points and weights) for integration over triangles. The raw data is available for electronic downloading from *arXiv.org*; see [75]. The same data, wrapped into a C program, is available at this book's website as files *twb-quad.[ch]*. In this section I will describe the organization of the data in those files and explain how to use them in computations. I call the resulting procedure *TWB quadrature* after the initials of the authors. Before we go on, I must add that there are other data sets for quadrature on triangles. Lyness and Cools [42] provide a survey of the available methods at the time of its publication in 1994. This was updated later in Cools [13]. Wandzura and Xiao [78] provide an alternative set of quadrature tables. The article by Xiao and Gimbutas [82] generalizes the technique to higher-dimensional domains. Also see the references in [74]. My choice of the TWB quadrature for the purposes of this book is mostly due to the easy availability of their data.

The TWB quadrature data consists of 14 tables, each corresponding to a certain *quadrature strength*. A quadrature table is said to have a strength of d if it produces *exact answers* for all polynomials of degree less than or equal to d. The "exact answer" should be qualified, however, since it is subject to the computer's floating point roundoff errors, as well as the numerical precision of the tables themselves. The order of magnitude of the relative error in a TWB quadrature is roughly around 10^{-14}. Table 5.1 of [75] gives more detailed error estimates.

Table 24.1 lists the quadrature strengths, d, and the corresponding number of quadrature points, n, of the 14 TWB quadrature tables. Thus, for instance, a polynomial of degree 5 may be integrated *exactly* through a 10-point TWB quadrature. Table 24.2 lists that quadrature's 10 coordinates and weights. For each quadrature point it gives two of the three barycentric coordinates $(\lambda_1, \lambda_2, \lambda_3)$. The third may be computed from $\lambda_1 + \lambda_2 + \lambda_3 = 1$. The weights add up to 2, indicating that the area of the standard reference triangle, $|T_{\text{std}}|$, is 2. Integrating a function over a triangle is a matter of plugging the numbers from that table into the formula (24.1). Figure 24.2 shows the distribution of the TWB quadrature points corresponding to strengths $d = 5$ and $d = 11$.

Table 24.1: Here is the list of the quadrature strengths, d, and the corresponding number of quadrature points, n, of the 14 TWB quadrature tables. A polynomial of degree 5, for instance, may be integrated *exactly* through a 10-point quadrature.

d	2	4	5	7	9	11	13	14	16	18	20	21	23	25
n	3	6	10	15	21	28	36	45	55	66	78	91	105	120

Table 24.2: The TWB data for a quadrature of strength $d = 5$.

	λ_1	λ_2	w
1	0.0000000000000	1.0000000000000	0.0262712099504
2	1.0000000000000	0.0000000000000	0.0262716612068
3	0.0000000000000	0.0000000000000	0.0274163947600
4	0.2673273531185	0.6728199218710	0.2348383865823
5	0.6728175529461	0.2673288599482	0.2348412238268
6	0.0649236350054	0.6716530111494	0.2480251793114
7	0.6716498539042	0.0649251690029	0.2480304922521
8	0.0654032456800	0.2693789366453	0.2518604605529
8	0.2693767069140	0.0654054874919	0.2518660533658
10	0.3386738503896	0.3386799893027	0.4505789381914

24.3 ▪ The files *twb-quad.[ch]*

I have wrapped the raw TWB quadrature data provided in [75] into C structures and placed them in the files *twb-quad.[ch]*, which you should download from the book's website. I will not describe the contents of *twb-quad.c*—I expect that you will use it as a "black box". Feel free to examine its contents if you wish; it's commented extensively.

What you should really understand is the interface, *twb-quad.h*, which is shown in Listing 24.1. I have stripped off the comments in that listing to save space. The actual file that you will download contains a good deal of comments.

Let us examine the contents of *twb-quad.h*:

Line 3. The preprocessor macro `TWB_STANDARD_AREA` defines the area of the standard reference triangle T_{std}, described earlier. Use that macro instead of hard-coding the number `2.0` in your programs.

Line 4. The structure declared here is designed to hold the data for one TWB quadrature point. That consists of the point's barycentric coordinates λ_1 and λ_2 and the corresponding weight w. The λ_3 coordinate is not in the TWB tables; our program calculates it from $\lambda_3 = 1 - \lambda_1 - \lambda_2$. An n-point quadrature table is an array of length n of such structures. See, however, Example 24.2 below.

Line 10. The function `twb_qdat()` returns a pointer to the TWB quadrature table of a requested strength. For instance, to get the quadrature table of strength $d = 10$, we do

```
int d = 10, n;
struct TWB_qdat *qdat = twb_qdat(&d, &n);
```

Here is how it works. We pass the *address*, not the value, of the variable `d` to `twb_qdat()`. This allows `twb_qdat()` to change the value of `d` if need be. This flexibility is necessary since the TWB quadrature provides tables for 14 strengths

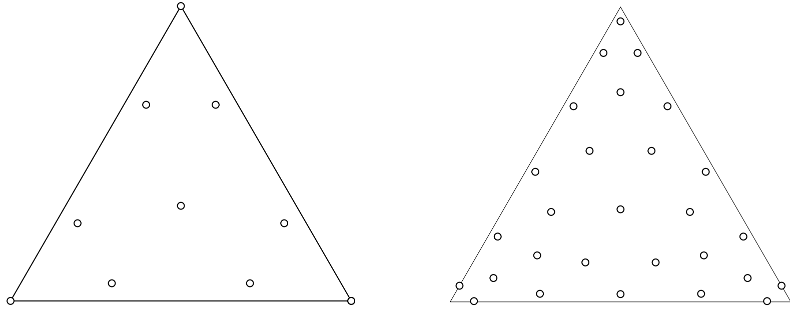


Figure 24.2: The distributions of the quadrature points in TWB quadratures of strengths $d = 5$ ($n = 10$ points) and $d = 11$ ($n = 28$ points). Note the asymmetric distribution of the points in the latter case. The article [75] has similar drawings for each of the 14 cases.

Listing 24.1: The file *twb-quad.h* provides an interface to the TWB quadrature tables.

```
1  #ifndef H_TWB_QUAD_H
2  #define H_TWB_QUAD_H
3  #define TWB_STANDARD_AREA 2.0
4  struct TWB_qdat {
5      double lambda1;
6      double lambda2;
7      double weight;
8      double lambda3;
9  };
10 struct TWB_qdat *twb_qdat(int *d, int *n);
11 #endif /* H_TWB_QUAD_H */
```

only (see Table 24.1), and therefore requests for arbitrary `d` cannot be granted. If the requested strength is not available, the next higher strength is selected. If the requested strength is greater than the maximum available, the maximum is selected. If it is smaller than the minimum available, the minimum is selected. In any case, the value of the variable `d` is set to the selected strength.

The function's second argument, `n`, is set to the number of quadrature points of the selected strength. For instance, if `d = 10`, as in the code fragment shown above, then after `twb_qdat()` returns, `d` is set to 11 and `n` is set to 28.

The function returns the selected quadrature table as a pointer to a (static) array of "`struct TWB_qdat`" of length `n`. Thus, in the code fragment shown above, the table's `i`th entry is produced by `qdat[i]`. The individual components of that entry are `qdat[i].lambda1`, `qdat[i].lambda2`, `qdat[i].lambda3`, and `qdat[i].weight`.

This completes the description of the details of *twb-quad.h*. You should be able to access and use the quadrature data in any way you wish. Let us look at a few examples.

Example 24.1. To print the table of strength $d = 10$, you may do

```
int d = 10, n, i;
struct TWB_qdat *qdat = twb_qdat(&d, &n);
```

```
printf("we have d = %d, n = %d\n", d, n);
for (i = 0; i < n; i++)
    printf("%3d: %15.12f %15.12f %15.12f %15.12f\n",
        i + 1, qdat[i].lambda1, qdat[i].lambda2, qdat[i].lambda3,
        qdat[i].weight);
```

Actually, this prints the table of strength $d = 11$ since no table of strength $d = 10$ exists.

Example 24.2. Here I will let you in on a little secret. In *twb-quad.c*, to each of the 14 quadrature tables I have appended a dummy quadrature point with the data $\lambda_1 = 0$, $\lambda_2 = 0$, and $w = -1$ that serves as a sentinel to mark the end of that table. This cannot be confused with a true quadrature point since the weight is negative while the TWB quadrature weights are all positive. Moreover, the call `twb_qdat(&d,&n)` sets `n` to the *true* number of the table's quadrature points (that is, it does not count the dummy point). We relied on that in the previous example. However, the presence of the end-marker provides an alternative way to read the table that can result in a more expressive code. Here is the previous example done in the new way:

```
int d = 10, n;
struct TWB_qdat *qdat = twb_qdat(&d, &n);
printf("we have d = %d, n = %d\n", d, n);
while (qdat->weight != -1) {
    printf("%15.12f %15.12f %15.12f %15.12f\n",
        qdat->lambda1, qdat->lambda2, qdat->lambda3, qdat->weight);
    qdat++;
}
```

Example 24.3. We saw in Example 24.2 that the scanning of a quadrature table may be done without a reference to the number n of the quadrature points. If you truly have no need for n, then you may call `twb_qdat()` with the second argument set to `NULL`, as in

```
int d = 10;
struct TWB_qdat *qdat = twb_qdat(&d, NULL);
printf("we have d = %d\n", d);
while (qdat->weight != -1) {
    printf("%15.12f %15.12f %15.12f %15.12f\n",
        qdat->lambda1, qdat->lambda2, qdat->lambda3, qdat->weight);
    qdat++;
}
```

Example 24.4. Listing 24.2 shows the contents of the file *twb-quad-minimal.c* that provides a bare-bones demonstration of the TWB quadrature. It integrates the function $f(x,y) = x^2 + y^2$ on the triangle T with vertices at $(0,0)$, $(1,0)$, and $(0,1)$ in the xy plane. The code is self-explanatory; therefore I will not elaborate on its details other than noting that on line 27 we multiply the sum by the factor $|T|/|T_{\text{std}}|$, in accordance with the formula (24.1). The area of our triangle T is 0.5.

Compile the program, and link with *twb-quad.o*. The exact value of the integral is $1/6$. Since the integrand is a polynomial of degree 2, applying the program with $d = 2$ should suffice to obtain the result within the accuracy provided by the TWB tables. Here is what I get:

Listing 24.2: The contents of the file *twb-quad-minimal.c*. This program integrates the function $f(x,y) = x^2 + y^2$ over the triangle T with vertices at $(0,0)$, $(1,0)$, and $(0,1)$.

```
1  #include <stdio.h>
2  #include "twb-quad.h"
3
4  double f(double x, double y)
5  {
6      return x*x + y*y;
7  }
8
9  int main(void)
10 {
11     double v1[] = { 0, 0 };      // vertex v1
12     double v2[] = { 1, 0 };      // vertex v2
13     double v3[] = { 0, 1 };      // vertex v3
14     double area = 0.5, sum = 0.0;
15     int d = 2, n, i;
16     struct TWB_qdat *qdat = twb_qdat(&d, &n);
17     printf("integrating with d = %d, n = %d\n", d, n);
18     for (i = 0; i < n; i++) {
19         double lambda1 = qdat[i].lambda1;
20         double lambda2 = qdat[i].lambda2;
21         double lambda3 = qdat[i].lambda3;
22         double w = qdat[i].weight;
23         double x = lambda1*v1[0] + lambda2*v2[0] + lambda3*v3[0];
24         double y = lambda1*v1[1] + lambda2*v2[1] + lambda3*v3[1];
25         sum += w*f(x,y);
26     }
27     sum *= area/TWB_STANDARD_AREA;
28     printf("integral = %.14f\n", sum);
29     return 0;
30 }
```

```
$ ./twb-quad-minimal
integrating with d = 2, n = 3
integral = 0.16666666666666
```

Remark 24.1. In accordance with the idea suggested in Example 24.2, the **for**-loop in *twb-quad-minimal.c* may be replaced with an index-free loop, as in

```
while (qdat->weight != -1) {
    double lambda1 = qdat->lambda1;
    double lambda2 = qdat->lambda2;
    double lambda3 = qdat->lambda3;
    double w = qdat->weight;
    double x = lambda1*v1[0] + lambda2*v2[0] + lambda3*v3[0];
    double y = lambda1*v1[1] + lambda2*v2[1] + lambda3*v3[1];
    sum += w*f(x,y);
    qdat++;
}
```

I prefer this this alternative method—it's a matter of personal taste—therefore that's what we will see in most of the TWB applications in the rest of this book.

Listing 24.3: The transcript of an interactive session with the program *twb-quad-demo*.

```
$ ./twb-quad-demo
Usage:  ./twb-quad-demo d a
    d = the TWB quadrature strength
    a = maximal triangle area

$ ./twb-quad-demo 4 0.01
quadrature strength = 4,  number of points = 6

domain is a triangle with hole
vertices = 158, edges = 408, elems = 250
geomview output written to file triangle_with_hole.gv
integral = -0.068572048611105

domain is an annulus (really a 12-gon) of radii 0.325 and 0.65
vertices = 94, edges = 246, elems = 152
geomview output written to file annulus.gv
integral = -2.9553796953088e-10

domain is a square
vertices = 96, edges = 254, elems = 159
geomview output written to file square.gv
integral = 0.9999999999999

domain is an L-shape with three holes (actually 12-gons
vertices = 112, edges = 274, elems = 160
geomview output written to file three_holes.gv
integral = 0.28554098238704
```

24.4 ▪ The program

With the help of the triangulation tools developed in Chapter 23, we are now in a position to integrate a function on any polygonal domain. It's a matter of triangulating the domain, integrating over the individual triangles, and adding the results. This capability is of utmost importance—integrating over arbitrary polygonal domains is at the heart of the FEM, as we will see in Chapter 25.

In this chapter's *Projects* section we write a program to compute $\iint_\Omega f(x,y)\,dx\,dy$ for various functions f over the domains Ω that we worked with in Chapter 23. As noted in Section 24.1, we are not going to encapsulate this chapter's work in a module since we will find it more practical to apply the *ideas and techniques*, rather than the programs, that we will develop here. Listing 24.3 is the transcript of an interactive session with this chapter's program.

As we see, the program is invoked as "`./twb-quad-demo 4 0.01`". It integrates four functions over the domains *triangle_with_hole*, *annulus*, *square*, and *three_holes*, which were introduced in Chapter 23, by triangulating the domains with a maximal area parameter of 0.01 and integrating over each triangle with a quadrature strength of 4. The integrands vary from case to case—they are specified in *problem-spec.c*. The integration over *square* computes

$$\int_0^1\int_0^1 36xy(1-x)(1-y)\,dx\,dy.$$

The exact value is 1, as it's trivial to evaluate that integral by hand. The program produces 0.9999999999999, which is within the floating point accuracy of the quadrature tables. An accurate outcome is expected since the integrand is a polynomial of degree 4 and we are applying a quadrature of strength 4.

The integration over *triangle_with_hole* computes

$$\iint_\Omega \frac{2}{5} xy(2-x)(2-y)\,dx\,dx.$$

The exact answer $-15799/230400 \approx -0.068572048611111\ldots$. Again, we see that the value computed by the program is within the floating point accuracy of the quadrature tables.

The integration over *annulus* computes

$$\iint_\Omega \frac{1}{5}\cos\left(3\tan^{-1}\frac{y}{x}\right)dx\,dy,$$

which is equivalent to integrating $\frac{1}{5}\cos 3t$ in polar coordinates. The integral's exact value is zero due to the symmetries of the domain and the integrand; however, we don't expect an exact quadrature since (a) the function being integrated is not a polynomial, and (b) the domain is bounded by polygons rather than true circles. Nonetheless, the computed result is quite respectably close to zero, albeit as a result of cancellations due to symmetries rather than a precise quadrature.

The integrand in the case of *three_holes* is $f(x,y) = x^2 + y^2$. You may change that if you wish.

Listing 24.4 shows the transcript of a sequence of additional experiments on *square* and the same fourth degree polynomial as before. I have set the maximal area parameter to 1. This produces the coarsest possible triangular mesh—two triangles only. In the three experiments shown, I reduce the quadrature strength from 4 to 3 to 2.

The result of the experiment on lines 1–6 is within 10^{-14} of the exact answer, as it should be, since we are integrating a fourth degree polynomial with quadrature of strength $d = 4$.

At first sight the outcome of the experiment on lines 8–13 seems unexpected since the answer remains unchanged despite the reduction in the quadrature strength. A closer look reveals, however—see line 9—that the program is applying a quadrature strength of 4 despite the request for a strength of 3. Why? See the list of the 14 available strengths in Table 24.1. *There is no table of strength $d = 3$.* The program detects that and picks up the next higher strength, which is $d = 4$.

On the other hand, the request for a quadrature strength of $d = 2$ on line 15 meets the program's approval and it duly produces the answer 0.9722, which shows a significant deterioration of accuracy, as may be expected since we are applying a quadrature of strength 2 to integrate a polynomial of degree 4.

24.4.1 ▪ Program files

In addition to the files *twb-quad.[ch]* discussed earlier, the implementation of the program depends on the files *xmalloc.[ch]* from Chapter 7, the file *array.h* from Chapter 8, and the files *mesh.[ch]* from Chapter 23. Furthermore, you will need a *copy* (not a link since we are going to modify it) of the file *problem-spec.c* from Chapter 23. The file *problem-spec.h* is not going to change; therefore a symbolic link is fine. Therefore, following the suggestions in

Listing 24.4: The transcript of a second interactive session with the program *twb-quad-demo*. We reduce the quadrature strength from 4 to 3 to 2. The mesh consists of two triangles only.

```
1   ./twb-quad-demo 4 1
2   quadrature strength = 4,  number of points = 6
3   domain is a square
4   vertices = 4, edges = 5, elems = 2
5   geomview output written to file square.gv
6   integral = 0.999999999999831
7
8   ./twb-quad-demo 3 1
9   quadrature strength = 4,  number of points = 6
10  domain is a square
11  vertices = 4, edges = 5, elems = 2
12  geomview output written to file square.gv
13  integral = 0.999999999999831
14
15  ./twb-quad-demo 2 1
16  quadrature strength = 2,  number of points = 3
17  domain is a square
18  vertices = 4, edges = 5, elems = 2
19  geomview output written to file square.gv
20  integral = 0.972222222222193
```

Chapters 2 and 6, your project's directory initially will look like this:[88]

```
$ cd twb-quad
$ ls -F
Makefile  plot-with-geomview.c  triangle.h@          twb-quad.c
array.h@  plot-with-geomview.h  triangle.o@          twb-quad.h
mesh.c@   problem-spec.c        twb-quad-demo.c      xmalloc.c@
mesh.h@   problem-spec.h@       twb-quad-minimal.c   xmalloc.h@
```

We saw the purpose and contents of *twb-quad-minimal.c* in Example 24.4 on page 324. In the following sections I will describe the contents of the files *plot-with-geomview.[ch]* and *twb-quad-demo.c* and delineate the required changes to *problem-spec.c*.

24.5 ▪ The files *plot-with-geomview.[ch]*

The integral of a function over a domain is just a number, and that's what we expect an integration program to produce. Our program, however, does more. In addition to producing the numerical value of the integral, it also writes a file—a *Geomview* script, to be precise—which may be fed to *Geomview* (see page 8) to plot a graph of the integrand as a surface in the three-dimensional space. The graphs in Figure 24.3 (page 335) are screenshots of *Geomview*'s display.

For the purposes of this chapter, plotting graphs is a mere frill and certainly not central to evaluating integrals. We do it to satisfy our curiosity regarding the shape of the domain

[88] Just as in Chapter 23, the files *triangle.[ho]* link to where you keep your *Triangle* files. If you have installed *Triangle* as a library, then these are not needed; just link your program with the `-ltriangle` flag instead. See Section 23.7 for more on this subject.

Listing 24.5: The file *plot-with-geomview.h* provides the interface to a module for writing *Geomview* scripts to visualize three-dimensional graphs of function defined on a mesh.

```
1 #ifndef H_PLOT_WITH_GEOMVIEW_H
2 #define H_PLOT_WITH_GEOMVIEW_H
3 #include "mesh.h"
4 void plot_with_geomview_mono(struct mesh *mesh, char *filename);
5 void plot_with_geomview_zhue(struct mesh *mesh, char *filename);
6 #endif /* H_PLOT_WITH_GEOMVIEW_H */
```

and the integrand's graph. If you are not interested in seeing the graph, you may safely remove the couple of lines of code that produce the *Geomview* script.

Plotting functions *will* be essential when we turn to FEM applications in Chapter 25. The only reason for including plotting code in this chapter is to prepare the way for future applications to the FEM.

The file *plot-with-geomview.c* contains the implementation of a module for writing *Geomview* scripts to plot functions over meshed domains., The module's interface, *plot-with-geomview.h*, is shown in Listing 24.5. The two functions declared on lines 4 and 5 produce monochrome and color graphics, respectively. You may call one or the other, or both, in any given situation. These functions receive the address of a mesh structure and a file name. They translate the contents of the structure to a format understandable to *Geomview* and write the result to the named file. If the file does not exist, it is created. If a file by that name exists, it will be overwritten by the new one.

The (x, y, z) coordinates of the graph's points are read from the mesh's node structures; see **struct** node in Listing 23.6 on page 312. You are responsible for calculating the z values—by evaluating the function to be plotted at every node—before passing the mesh structure to `plot_with_geomview_*()`.

Writing *plot-with-geomview.[ch]* requires some familiarity with *Geomview*'s syntax; therefore I am not making it a requirement for this project. Download those files from the book's website, and compile them along with the rest of your program.

24.6 ▪ Modifying the file *problem-spec.c*

We are going to modify slightly the file *problem-spec.c* that you copied here from Chapter 23's *Project Triangulate*. The modifications are simple. In that chapter we were concerned with triangulation only; therefore the member `f` of "**struct** problem_spec" was irrelevant and was set to `NULL`. In this chapter `f` points to a function that is to be integrated.

Thus, just before the definition of the function `triangle_with_hole()` (see Listing 23.3 on page 307) add a new function:

```
static double triangle_f(double x, double y)
{
    return 0.4*x*y*(2-x)*(2-y);
}
```

Then, on line 38 of Listing 23.3, change the `NULL` to `triangle_f`. Our program will integrate this function over the *triangle_with_hole* domain. The function and its name are arbitrary. Change them as you wish. Regardless of the name you give, the program will access that function through the member `f` of "**struct** problem_spec", as in `spec→f(x,y)`.

Listing 24.6: An outline of the file *twb-quad-demo.c*.

```
1  ▶ #include ...
2  ▶ static void eval_f(struct mesh *mesh,
3              double (*f)(double x, double y)) ...
4  ▶ static double integrate_over_triangle(struct elem *ep,
5              struct TWB_qdat *qdat,
6              double (*f)(double x, double y)) ...
7  ▶ static void do_demo(struct problem_spec *spec, double a,
8              struct TWB_qdat *qdat, char *gv_filename) ...
9  ▶ static void show_usage(const char *progname) ...
10 ▶ int main(int argc, char **argv) ...
```

Similarly, just before the definition of the function `annulus()` (see Listing 23.4 on page 310) add a new function,

```
static double annulus_f(double x, double y)
{
    double t = atan2(y,x);    // the polar angle of the point (x,y)
    return 0.20*cos(3*t);
}
```

then detach `spec→f` from line 40, and set `spec→f = annulus_f`.

24.7 ▪ The file *twb-quad-demo.c*

The file *twb-quad-demo.c* demonstrates the TWB quadrature technique by integrating functions over the various domains defined in *problem-spec.c*; see Listing 24.3 for a transcript of an interactive session. Listing 24.6 gives the file's outline. I will begin the description of the contents of the file *twb-quad-demo.c* at the bottom, that is, at the function `main()`, and work my way up.

24.7.1 ▪ The function `main()`

Listing 24.7 gives the contents of the function `main()`. Let us look at its details.

Lines 3–8. These supply the prototypes of the functions `square()`, `annulus()`, etc., which are defined in *problem-spec.c*. I could have included those prototypes in the file *problem-spec.h*, but I decided against it since adding new functions in *problem-spec.c* would necessitate changing *problem-spec.h*, which would not be pretty. Instead, I have chosen to include the prototypes in `main()` since that's the only place where they are needed.

Line 15. We expect the program to be invoked with two command-line arguments, as in "`./twb-quad-demo d a`", where `d` is the desired quadrature strength, and `a` is an upper bound on the triangle areas in the triangulation. Thus, if `argc` is other than 3, we print a usage message and exit the program.

Line 19. We call the C standard library's `strtol()` function (see Chapter 5) to extract the value of `d` from the command-line parameter `argv[1]`. If the user has supplied an illegal value, we print a usage message message and exit the program.

Listing 24.7: The function `main()` in the file *twb-quad-demo.c*.

```
int main(int argc, char **argv)
{
    struct problem_spec *square(void);
    struct problem_spec *triangle_with_hole(void);
    struct problem_spec *annulus(int n);
    void free_annulus(struct problem_spec *spec);
    struct problem_spec *three_holes(int n);
    void free_three_holes(struct problem_spec *spec);
    struct problem_spec *spec;
    struct TWB_qdat *qdat;
    int d;                    // the quadrature strength
    int n;                    // the number of quadrature points
    double a;                 // upper bound on triangle areas
    char *endptr;
    if (argc ≠ 3) {
        show_usage(argv[0]);
        return EXIT_FAILURE;
    }
    d = strtol(argv[1], &endptr, 10);
    if (*endptr ≠ '\0') {
        show_usage(argv[0]);
        return EXIT_FAILURE;
    }
    a = strtod(argv[2], &endptr);
    if (*endptr ≠ '\0' || a ≤ 0) {
        show_usage(argv[0]);
        return EXIT_FAILURE;
    }
    qdat = twb_qdat(&d, &n);
    printf("quadrature strength = %d,  number of points = %d\n", d, n);
    putchar('\n');

    do_demo(triangle_with_hole(), a, qdat, "triangle-with-hole.gv");
    putchar('\n');

    spec = annulus(12);
    do_demo(spec, a, qdat, "annulus.gv");
    free_annulus(spec);
    putchar('\n');
    ...
```

Line 24. We call the C standard library's `strtod()` function (see Chapter 5) to extract the value of a from the command-line parameter `argv[2]`. If the user has supplied an illegal value, we print a usage message message and exit the program.

Line 29. We call `twb_qdat()` to fetch the quadrature table of strength `d`. The value of `d` will be adjusted, if necessary, to conform to one of the 14 possible values; see Table 24.1 on page 322.

Line 33. We call `do_demo()` (more on that later) to perform an integration over the domain returned by `triangle_with_hole()` as defined in the file *problem-spec.c*.

Listing 24.8: The function `do_demo()` in the file *twb-quad-demo.c*.

```
1  static void do_demo(struct problem_spec *spec, double a,
2          struct TWB_qdat *qdat, char *gv_filename)
3  {
4      double sum = 0.0;
5      struct mesh *mesh = make_mesh(spec, a);
6      printf("vertices = %d, edges = %d, elems = %d\n",
7              mesh->nnodes, mesh->nedges, mesh->nelems);
8      eval_f(mesh, spec->f);
9      plot_with_geomview_mono(mesh, gv_filename);
10     for (int i = 0; i < mesh->nelems; i++)
11         sum += integrate_over_triangle(&mesh->elems[i], qdat, spec->f);
12     printf("integral = %.14g\n", sum);
13     free_mesh(mesh);
14 }
```

The program will integrate the function associated with the domain and print the result to the `stdout`. Additionally, it will write a three-dimensional representation of the integrand's graph into a file named in `do_demo()`'s last argument; it is `"triangle-with-hole.gv"` in this case. The name of the file is arbitrary. The graph may be viewed with *Geomview*. See Figure 24.3 on page 335 for samples.

Line 36. We integrate over the annulus. In contrast to the previous integration, here we separate the call to `annulus()` from the call to `do_demo()`. This is because we want to save a pointer, `spec`, to the structure returned by `annulus()` so that we may free the memory later by calling `free_annulus()`. This was not necessary in the case of `triangle_with_hole()` since that function does not allocate memory.

Line 40. I am truncating the listing of `main()` here. In this chapter's *Projects* section you will receive instructions on adding more demos here.

24.7.2 ▪ The function `show_usage()`

The function `show_usage()` that appears on line 9 of Listing 24.6 prints a brief usage message. See the transcript of a sample session in Listing 24.3. Write something similar for your program. Make the message more detailed if you wish.

24.7.3 ▪ The function `do_demo()`

The function `do_demo()` that appears on line 7 of Listing 24.6 runs the demo for one integration problem. It receives the problem specification in the `spec` argument, a pointer to the TWB quadrature table in the `qdat` argument, as well as the maximal area value of `a`, and a file name for writing a *Geomview* script for a three-dimensional rendering of the integrand's graph. Listing 24.8 shows the implementation of `do_demo()`. It is mostly self-explanatory, but nevertheless, let's look at some of its details.

Line 5. We pass the problem specification in `spec` and the value of maximal area parameter `a` to the function `make_mesh()` (from Chapter 23) to mesh the domain. After the function returns, the pointer `mesh` points to the resulting mesh structure.

Listing 24.9: The function `integrate_over_triangle()` in the file *twb-quad-demo.c*.

```
1  static double integrate_over_triangle(struct elem *ep,
2          struct TWB_qdat *qdat, double (*f)(double x, double y))
3  {
4      double x[3], y[3];
5      double sum = 0.0;
6      for (int i = 0; i < 3; i++) {
7          x[i] = ep->n[i]->x;
8          y[i] = ep->n[i]->y;
9      }
10     while (qdat->weight != -1) {
11         double lambda[3];
12         lambda[0] = qdat->lambda1;
13         lambda[1] = qdat->lambda2;
14         lambda[2] = qdat->lambda3;
15         double X = lambda[0]*x[0] + lambda[1]*x[1] + lambda[2]*x[2];
16         double Y = lambda[0]*y[0] + lambda[1]*y[1] + lambda[2]*y[2];
17         sum += qdat->weight * f(X,Y);
18         qdat++;
19     }
20     return (ep->area / TWB_STANDARD_AREA) * sum;
21 }
```

Line 8. The call to `eval_f()`, explained fully in subsection 24.7.5, evaluates the integrand, that is, `spec->f`, at every node of the mesh and stores the value in the node structure's `z` member; see line 4 of Listing 23.6 on page 312. This step is not necessary at all for the integration. It is done only for producing a three-dimensional drawing of the integrand's graph; see the next step.

Line 9. The function `plot_with_geomview_mono()` is defined in the file *plot-with-geomview.c*, which you downloaded from the book's website; see Section 24.5. It reads the mesh data and produces a script which may be fed to *Geomview* to produce a three-dimensional monochrome rendering of the integrand. The value of the integrand at each node is obtained from node structure's `z` member, as noted above. Replace the call with `plot_with_geomview_zhue()` to obtain a colored graph.

This step is not necessary at all for the integration. If you are not interested in seeing the integrand's graph, you may safely remove lines 8 and 9 from `do_demo()`.

Line 10. Here is where the real work is done. The **`for`**-loop goes over the mesh's elements (that is, triangles), calls `integrate_over_triangle()` (more about that later) to integrate over individual triangles, and accumulates the results. Afterward the accumulated value is printed and the memory associated with the mesh is freed.

24.7.4 ▪ The function `integrate_over_triangle()`

The function `integrate_over_triangle()` that appears on line 4 of Listing 24.6 is where the main action takes place. It integrates a function over a triangle through a TWB quadrature. Its implementation is shown in its entirety in Listing 24.9.

The function's first argument, `ep`, is a pointer to a **`struct`** `elem` (that is, a triangle) declared in the file *mesh.h*; see line 16 of Listing 23.6 on page 312. The function's second argument is a pointer to a TWB quadrature table, and the third argument is a function of the type $\boldsymbol{R}^2 \to \boldsymbol{R}$. The goal is to integrate that function over the triangle and return the result. Let us examine the details.

Line 6. The expression `ep→n[i]` is a pointer to the node at the element's `i`th vertex. Therefore the vertex's coordinates are `ep→n[i]→x` and `ep→n[i]→y`. These expressions are rather clumsy; therefore we introduce the shorthands `x[i]` and `y[i]` for these.

Line 10. The **`while`**-loop goes over the quadrature points. It stops when we reach the quadrature table's end-marker, which is a point with weight −1 (see Example 24.2 on page 324). We introduce `lambda[i]` as a shorthand for the barycentric coordinates `qdat→lambda1`, etc., and then calculate the Cartesian coordinates (X, Y) of the quadrature point in accordance with (A.1) on page 375. Then we evaluate the function f at the quadrature point, multiply the result by the corresponding weight, and accumulate the results in the variable `sum`. In effect, this is the C implementation of the summation in formula (24.1) on page 320.

Line 20. This corresponds to the multiplication by $|T|/|T_{\text{std}}|$ in formula (24.1).

24.7.5 ▪ The function `eval_f()`

The function `eval_f()` that appears on line 2 of Listing 24.6 receives pointers to a mesh and a function. It evaluates the function at every node and places the result in the node structure's `z` member. The computation is quite straightforward, so I will let you write the code.

24.8 ▪ *Project TWB Quadrature*

Part 24.1. You have all the bits and pieces needed to compile and test the TWB quadrature program. I suggest that you begin with *twb-quad-minimal.c* presented in Example 24.4 to gain some assurance that things work as they should. Then proceed with the various items below.

Part 24.2. Complete the loose ends. Based on Section 24.6's narrative, the domain-specification functions `triangle_with_hole()` and `annulus()` and the associated functions (integrands) `triangle_f()` and `annulus_f()` should be ready to go. Compile, link, and execute the program. Compare its output to that shown in the transcript of the interactive session in Listing 24.3.

Part 24.3. In *problem-spec.c* you have a function `square()` that defines a domain in the form of the unit square $[0,1] \times [0,1]$. Add the associated function `square_f()` as $f(x.y) = 36xy(1-x)(1-y)$ whose exact integral over the square is 1. Verify that your program confirms that. You will need to add code at the bottom of `main()` in the file *twb-quad-demo.c* to activate this demo.

Part 24.4. Change `square_f()` to $f(x,y) = \frac{40}{3}x(1-x^2)y(1-y^3)$. (Don't delete the previous function; just comment it out.) This also integrates to 1 on $[0,1] \times [0,1]$. What is a good quadrature strength for it?

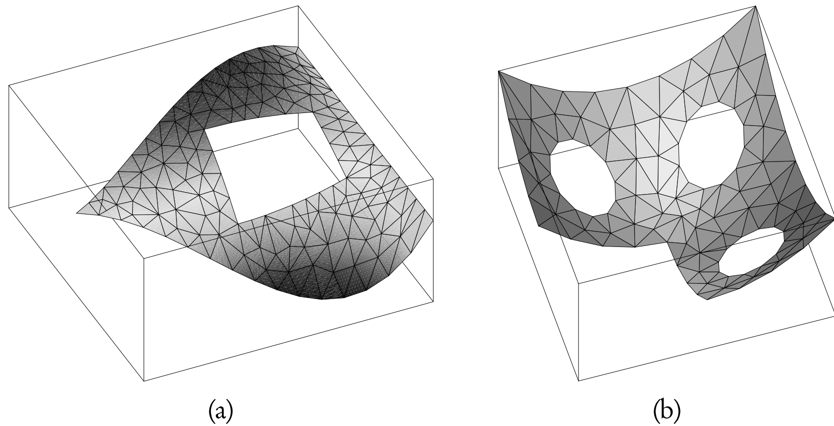


Figure 24.3: The graphs of the integrands of the problem specifications in `triangle_with_hole()` and `three_holes()`, produced by executing "`./twb-quad-demo 4 0.01`" and then viewing the results in *Geomview*. The color figure on the left is made by `plot_with_geomview_zhue()` and then printed in greyscale on this page, and the monochrome one on the right is made by `plot_with_geomview_mono()`.

Part 24.5. Try a nonpolynomial for `square_f()`. The function $f(x,y) = \frac{\pi^2}{4} \sin \pi x \sin \pi y$ integrates to 1 over the square. What does your program say?

Part 24.6. In *problem-spec.c* you have a function `three_holes()` that defines an L-shaped domain with three holes; see Figure 23.1 on page 302. Add the associated function `three_holes_f()` as $f(x,y) = x^2 + y^2$, and extend `main()` to activate the demo. Compare the result with that shown in Listing 24.3. View the graphics file that the program generates. Mine is shown in Figure 24.3(b).

Chapter 25

Finite elements

Prerequisites: Chapters 7, 8, 11, 12, 23, 24, and Appendix A

25.1 ▪ The Poisson equation

The simplest introduction to the *finite element method* (FEM) for solving partial differential equations (PDEs) is through the *Poisson equation*[89] in two dimensions:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + f = 0, \tag{25.1}$$

where $f = f(x,y)$ is given, and $u = u(x,y)$ is unknown. You can see right away that a solution u, if one exists, is not unique. For instance, if $u(x,y)$ is a solution, then $u(x,y) + ax + by + c + e^{\alpha x}\sin\alpha y + e^{\beta x}\cos\beta y$ with arbitrary constants a, b, c, α, and β also is a solution since those extra terms go away when you differentiate them twice and add. The slack in determining $u(x,y)$ is much wider than the five extra terms that I have shown. Any member of the rich class of functions known as *harmonic functions* is a part of that slack.

Practical applications supplement the Poisson equation with additional data that help to single out a unique solution out of the infinitely many candidates. Consider, for example, a prescribed domain $\Omega \in \boldsymbol{R}^2$ and a given function $f : \Omega \to \boldsymbol{R}$, and pose the following *boundary value problem*: Find $u : \Omega \to \boldsymbol{R}$ such that[90]

$$\begin{aligned} \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + f &= 0 \quad \text{in } \Omega, \\ u &= 0 \quad \text{on } \partial\Omega. \end{aligned} \tag{25.2}$$

The boundary value problem (25.2) occurs in innumerable contexts in mathematical models in physics and engineering. Almost every elementary book on PDEs and every book in the "mathematical methods in physics/engineering" genre will give the derivation of the boundary value problem (25.2), or variants thereof, in numerous contexts,

[89]The equation is named after the French mathematician Siméon Denis Poisson, pronounced *pwa-son*. The three-dimensional version of the equation is $\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} + f = 0$, where u and f are functions of x, y, and z.

[90]The notation "$\partial\Omega$" is a conventional way of referring to the boundary of Ω. The "∂" there has nothing to do with partial derivatives.

such as heat conduction, equilibrium of membranes under applied loads, and distribution of charges on electrically conducting objects. The books [38, 81, 20, 43, 79, 83] are but a small sample of the titles in this vast area.

It turns out that if (a) the geometry of Ω is nice, e.g., it is bounded and has a nonempty interior; (b) $\partial\Omega$ has no severe kinks or weird things like fractals; and (c) the function f is nice, e.g., it does not have severe discontinuities, then the boundary value problem (25.2) has a unique solution u. The FEM may be applied to find an approximation to that u with a desired level of accuracy. It is the goal of this chapter to learn how.

25.2 ▪ The weak formulation

25.2.1 ▪ Gradient, divergence, and Laplacian

The *gradient* of a function $u : \boldsymbol{R}^n \to \boldsymbol{R}$ is the vector $\nabla u = \langle \frac{\partial u}{\partial x_1}, \frac{\partial u}{\partial x_2}, \dots, \frac{\partial u}{\partial x_n} \rangle$, where $x_1, x_2, \dots, x_n$ are the Cartesian coordinates. Insofar as ∇u is the list of the first order partial derivatives of u, it does not matter whether you think of it as a row or column vector. However, when ∇u appears in the context of matrix algebra, it is *always* regarded as a *column vector* so that an expression of the sort $A\nabla u$, where A is an $n \times n$ matrix, makes sense.

The *gradient operator* is the symbolic vector $\nabla = \langle \frac{\partial}{\partial x_1}, \frac{\partial}{\partial x_2}, \dots, \frac{\partial}{\partial x_n} \rangle$. One thinks of ∇u as the operator ∇ acting on the function u. If $\boldsymbol{v}$ is a vector field, that is, $\boldsymbol{v} : \boldsymbol{R}^n \to \boldsymbol{R}^n$, then ∇ can operate on $\boldsymbol{v}$ as in a dot product:

$$\nabla \cdot \boldsymbol{v} = \Big\langle \frac{\partial}{\partial x_1}, \frac{\partial}{\partial x_2}, \dots, \frac{\partial}{\partial x_n} \Big\rangle \cdot \langle v_1, v_2, \dots, v_n \rangle = \frac{\partial v_1}{\partial x_1} + \frac{\partial v_2}{\partial x_2} + \dots + \frac{\partial v_n}{\partial x_n}.$$

The expression $\nabla \cdot \boldsymbol{v}$ occurs frequently in applications, especially in continuum mechanics and electromagnetism. It is called the *divergence of the vector field* $\boldsymbol{v}$. (The notation $\operatorname{div} \boldsymbol{v}$ is another way of writing $\nabla \cdot \boldsymbol{v}$.) In particular, if $\boldsymbol{v}$ is a *potential vector field*, that is, $\boldsymbol{v} = \nabla u$, where u is a scalar function, then we get

$$\operatorname{div} \nabla u = \nabla \cdot \nabla u = \nabla \cdot \Big\langle \frac{\partial u}{\partial x_1}, \frac{\partial u}{\partial x_2}, \dots, \frac{\partial u}{\partial x_n} \Big\rangle = \frac{\partial^2 u}{\partial x_1^2} + \frac{\partial^2 u}{\partial x_2^2} + \dots + \frac{\partial^2 u}{\partial x_n^2}.$$

The result, which is really the divergence of the gradient of u, is called the *Laplacian* of u. Like the divergence, this too occurs in very many places in applications to mechanics and physics. Although $\nabla \cdot \nabla u$ is a perfectly good notation for the concept, you may see it written variously as $\nabla^2 u$ (engineers and physicists like that) or Δu (mathematicians like that). Accordingly, the Poisson equation (25.1) may be written variously as

$$\nabla \cdot \nabla u + f = 0 \quad \text{or} \quad \operatorname{div} \nabla u + f = 0 \quad \text{or} \quad \nabla^2 u + f = 0 \quad \text{or} \quad \Delta u + f = 0.$$

These are more compact than (25.1), and more general as well, since they are not limited to two dimensions.

25.2.2 ▪ The Divergence Theorem

The *Divergence Theorem* is an indispensable tool when working with PDEs. A first course in multivariable calculus is likely to present the Divergence Theorem only for the $n = 2$

and $n=3$ cases. In the $n=2$ case it is called *Green's Theorem* and is stated as

$$\iint_{\Omega} \operatorname{div} \boldsymbol{F}(x,y)\,da = \int_{\partial\Omega} \boldsymbol{F}\cdot\boldsymbol{n}\,ds,$$

where $\boldsymbol{F}:\Omega\subset\boldsymbol{R}^2\to\boldsymbol{R}^2$ is a vector field on the two-dimensional domain Ω and $\boldsymbol{n}$ is the unit outward normal to the boundary, $\partial\Omega$ (see, e.g., the section titled "Curl and Divergence" in Stewart [65]). In the $n=3$ case it is called the *Divergence Theorem* and is stated as

$$\iiint_{\Omega} \operatorname{div} \boldsymbol{F}(x,y,z)\,dv = \iint_{\partial\Omega} \boldsymbol{F}\cdot\boldsymbol{n}\,da,$$

where $\boldsymbol{F}:\Omega\subset\boldsymbol{R}^3\to\boldsymbol{R}^3$ is a vector field on the three-dimensional domain Ω and $\boldsymbol{n}$ is the unit outward normal to the boundary, $\partial\Omega$ (see, e.g., the section titled "The Divergence Theorem" in Stewart [65]).

A more advanced course in multivariable calculus is likely to present the Divergence Theorem on $\boldsymbol{R}^n$ for any n in the form

$$\int_{\Omega} \operatorname{div} \boldsymbol{F}\,dx = \int_{\partial\Omega} \boldsymbol{F}\cdot\boldsymbol{n}\,da, \tag{25.3}$$

where $\boldsymbol{F}:\Omega\subset\boldsymbol{R}^n\to\boldsymbol{R}^n$ is a vector field on the n-dimensional domain Ω and $\boldsymbol{n}$ is the unit outward normal to the boundary, $\partial\Omega$. The left-hand side represents an n-tuple integral over Ω. The right-hand side is a surface integral over Ω's boundary.

Although the implementations of FEM are limited to two-dimensional domains in this book, I will use the general form (25.3) of the Divergence Theorem in the discussion since it is less cumbersome than the others.

A very useful consequence of the Divergence Theorem (25.3) is obtained by setting $\boldsymbol{F}=v\nabla u$, where u and v are arbitrary (scalar) functions on $\Omega\subset\boldsymbol{R}^n$. We have

$$\operatorname{div}\boldsymbol{F}=\nabla\cdot(v\nabla u)=\sum_{i=1}^{n}\frac{\partial}{\partial x_i}\left(v\frac{\partial u}{\partial x_i}\right)=\sum_{i=1}^{n}\frac{\partial v}{\partial x_i}\frac{\partial u}{\partial x_i}+\sum_{i=1}^{n}v\frac{\partial^2 u}{\partial x_i^2}=\nabla u\cdot\nabla v+v\nabla\cdot\nabla u;$$

therefore, the Divergence Theorem implies that

$$\int_{\Omega}(\nabla u\cdot\nabla v+v\nabla\cdot\nabla u)\,dx=\int_{\partial\Omega}v\nabla u\cdot\boldsymbol{n}\,da=\int_{\partial\Omega}v\frac{\partial u}{\partial n}\,da. \tag{25.4}$$

This identity, which holds between any two scalar functions u and v on any domain $\Omega\subset\boldsymbol{R}^n$,[91] is called *Green's identity*.

25.2.3 ▪ The weak formulation

For convenience, I will reproduce the boundary value problem (25.2) here with the equivalent but more compact notation

$$\begin{aligned} \nabla\cdot\nabla u+f&=0 \quad \text{on } \Omega,\\ u&=0 \quad \text{on } \partial\Omega. \end{aligned} \tag{25.5}$$

Let V be the set of (smooth) functions defined on Ω that vanish on $\partial\Omega$. The solution u belongs to that set. Pick an arbitrary function $v\in V$. Plug v and the solution u of (25.5)

[91] I am glossing over technical issues such as differentiability and integrability of the functions and the smoothness of the domain.

into Green's identity in (25.4). The boundary integral drops out since v is zero on the boundary, and we are left with $\int_\Omega (\nabla u \cdot \nabla v - vf)\,dx = 0$, which we rearrange as

$$\int_\Omega \nabla u \cdot \nabla v \, dx = \int_\Omega f v \, dx. \tag{25.6}$$

We conclude that a solution u of the boundary value problem (25.5) satisfies the identity (25.6) for every possible function v in V. It can be shown that the converse is also true: If (25.6) holds for a function $u \in V$ and all functions $v \in V$, then u is a solution of (25.5). In those terms, *the boundary value problem* (25.5) *and the identity* (25.6) *are equivalent.* The identity (25.6) is called the *weak formulation of the boundary value problem* (25.5). In light of this, solving the boundary value problem (25.5) is equivalent to the following statement:

Find $u \in V$ such that

$$\int_\Omega \nabla u \cdot \nabla v \, dx = \int_\Omega f v \, dx \quad \text{for all } v \in V. \tag{25.7}$$

I am being intentionally evasive about the precise nature of the set V. If V is going to serve as a source of functions for u and v, then we had better impose some constraints on the types of functions that go into it. For instance, such functions need to be differentiable in some sense so that the expressions ∇u and ∇v in (25.7) have meanings. Additionally, such functions need to be integrable in some sense because otherwise the integrals in (25.7) won't have meanings. Furthermore, the set V should be sufficiently rich; otherwise the weak formulation (25.7) can't be equivalent to the boundary value problem (25.5). The precise characterization of V involves the Sobolev space $H^1(\Omega)$ and its subspaces. I will not elaborate on these here. See the references at the end of this chapter for further study. I have strived to present the FEM in this chapter in such a way that it should remain accessible, at least in a general outline, even if you don't grasp the technical details. A lack of knowledge of the Sobolev spaces, for instance, should not hinder the completion of this chapter's projects.

Remark 25.1. There is magic in (25.7). The PDE in (25.5) is of second order since u is being differentiated twice. In the slight of hand that leads to (25.7), the second order derivatives disappear. That effect has wide-ranging repercussions in the theory of PDEs.

25.3 ▪ The Galerkin approximation

Once the vague ideas introduced in the previous section are placed on a firm mathematical foundation, it emerges that the set V of functions introduced in the previous section is a *linear space*, that is, if f and g are in V, then $\alpha f + \beta g$ is in V for all constants α and β. Furthermore, the linear space V is *infinite dimensional*; there is no finite basis that spans V. The *Galerkin approximation*[92] amounts to replacing the infinite-dimensional space V with an m-dimensional subspace, say $V_m \subset V$, and then reformulating (25.7) as follows:

[92] It is named after the Russian engineer/mathematician Boris Galerkin, the proper romanization of whose name is *Galyorkin*, but in English-speaking countries it is generally spelled and pronounced “Galerkin”.

Find $u_m \in V_m$ such that

$$\int_\Omega \nabla u_m \cdot \nabla v \, dx = \int_\Omega f v \, dx \quad \text{for all } v \in V_m. \tag{25.8}$$

One wants V_m to approximate V well enough so that the solution u_m of (25.8) is near the solution u of (25.7) if m is large. We will look at the construction of such a V_m in the next section. For now, suppose that we have succeeded in constructing a basis $\mathcal{B}_m = \{v_i\}_{i=1}^m$ for V_m. Then the condition "for all $v \in V_m$" in (25.8) is equivalent to "for $v = v_i$, $i = 1, 2, \ldots, m$". Furthermore, any $u_m \in V_m$ may be expressed as a linear combination of V_m's basis functions:

$$u_m = \sum_{j=1}^m c_j v_j. \tag{25.9}$$

Consequently, the equation in (25.8) takes the form

$$\int_\Omega \nabla\Big(\sum_{j=1}^m c_j v_j\Big) \cdot \nabla v_i \, dx = \int_\Omega f v_i \, dx, \quad i = 1, 2, \ldots, m,$$

which, upon a slight rearrangement of terms, leads us to the following equivalent formulation of Galerkin's approximation:

Find c_j, $j = 1, 2, \ldots, m$, such that

$$\sum_{j=1}^m \Big(\int_\Omega \nabla v_i \cdot \nabla v_j \, dx\Big) c_j = \int_\Omega f v_i \, dx, \quad i = 1, 2, \ldots, m. \tag{25.10}$$

At this point we introduce an $m \times m$ matrix K and an m-vector F through

$$K_{ij} = \int_\Omega \nabla v_i \cdot \nabla v_j \, dx, \qquad F_i = \int_\Omega f v_i \, dx, \tag{25.11}$$

and an m-vector c with components $c = \langle c_1, c_2, \ldots, c_m \rangle$, whereupon Galerkin's approximation (25.10) takes the form of a linear system of m equations in m unknowns:

$$Kc = F. \tag{25.12}$$

Thus, finding the approximate solution u_m of the boundary value problem (25.5) has been reduced to computing the integrals in (25.11), solving the linear system (25.12) for c, and then evaluating the sum (25.9).

The $m \times m$ matrix K is called the *system stiffness matrix* and the vector F is called the *system force vector*, terminology that dates back to the early years of the FEM and its use in solving elasticity problems.

25.4 ▪ An overview of the FEM

To begin implementing the previous section's Galerkin approximation idea, we need to construct a basis $\mathcal{B}_m = \{v_i\}_{i=1}^m$ for the subspace V_m and compute the stiffness matrix K and the force vector F in (25.11). We will see how these are done in the following four subsections.

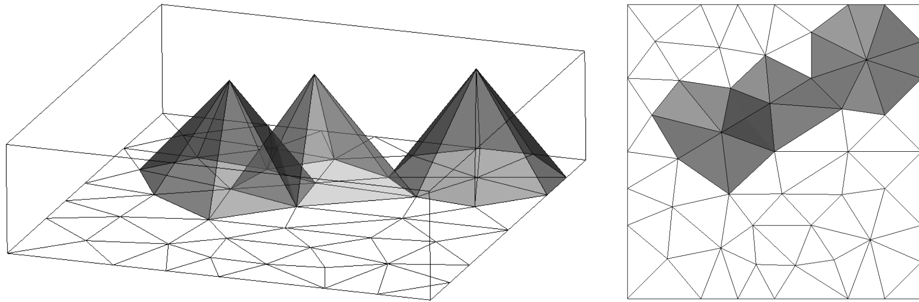


Figure 25.1: A triangulated square and a few basis functions, shown in perspective (left) and viewed from above (right). Note the overlap of the supports of the adjacent basis functions.

25.4.1 ▪ Meshing

The FEM's first step in the construction of the finite-dimensional subspace V_m and its basis $\mathscr{B}_m$ is to mesh the domain Ω. In this book we will consider triangular meshes only, although quadrilateral and hexagonal meshes are also in common use. Figure 25.1 shows a sample triangulated domain Ω which happens to be a square in this instance, but the fact that it is a square has no bearing on what follows. The figure also shows a few pyramids which are constructed as follows. Pick any vertex, say vertex number i, of the mesh. Identify all the triangles that have that vertex in common. Construct a pyramid of height 1, whose apex is situated directly above the selected vertex and whose base consists of the union of the triangles that you identified. Consider the pyramid as *the graph* of a function of two variables. Extend that function by zero to the rest of the domain Ω. Call the resulting function v_i. Repeat this for every vertex of the mesh other than those vertices that fall on the domain's boundary. This produces a collection of functions $\mathscr{B}_m = \{v_i\}_{i=1}^m$, where m is the number of the mesh's internal vertices. Each function $v_i \in \mathscr{B}_m$ is defined over the entire Ω, and has a graph in the form of a pyramid locally, and extended by zero globally. It should be clear that $\mathscr{B}_m$ is a linearly independent set—you cannot construct one $v_i \in \mathscr{B}_m$ as a linear combination of the others since v_i is 1 at the vertex i while all others are zero there.

Define the linear space V_m as the span of $\mathscr{B}_m$. Since $\mathscr{B}_m$ is a basis for V_m and $\mathscr{B}_m$ has m elements, we have $\dim V_m = m$. Refining the mesh increases the number of internal vertices and hence the dimension of V_m.

Every function in V_m vanishes on the boundary since we skipped the boundary vertices when constructing the functions v_i. It takes some quite technical work to show that V_m is a good approximation to V, and as m goes to infinity, the solution u_m of (25.8) approaches the solution u of (25.7). See any of [12, 11, 10, 23, 5] for the details.

25.4.2 ▪ Assembling the system

Now that we have constructed the pyramid-shaped basis functions v_i, we should be ready to compute the system stiffness matrix K and the system force vector F through the integration formulas (25.11). At first sight, computing K_{ij} seems to be straightforward: for every pair of vertices i and j, we look at the pyramids centered at i and j, multiply their gradients, and integrate. The integration takes place on the overlap region of bases of the pyramid pair since outside the overlap region at least one of the functions v_i and v_j is

zero. This gives an interesting structure to the system stiffness matrix: if the vertices i and j are not adjacent, then their pyramids do not overlap, and therefore K_{ij} is zero. In a fine mesh it's more likely than not that a randomly picked pair of vertices is not adjacent, and therefore most of the K matrix consists of zeros; that is, K is a *sparse matrix*.

The sketch above of the procedure for computing the system stiffness matrix K is *vertex-oriented*; in essence, it computes the entries K_{ij} through a doubly nested **`for`**-loop of the indices i and j which walk over the mesh's vertices. In principle one may implement such a computation, although I have never seen it done. The major impediment is the mess in determining the overlap region of the bases of a pair of pyramids. It's just not worth the trouble, especially since a much slicker alternative is available.

The alternative shifts the focus from *vertices* to *triangles*—instead of the doubly nested **`for`**-loop that walks over the vertices, we implement a single **`for`**-loop that walks over the mesh's triangles. Here is how it works.

We have already noted that the entry K_{ij} of the system stiffness matrix is the sum of the integrals over the cluster of triangles that forms the overlap region between the pyramids centered at the vertices i and j. Instead of focusing on such clusters, we compute integrals on every single triangle in the mesh, regardless of any considerations of to which cluster the triangle may belong. Only retroactively do we figure out where in the K matrix the result should be inserted. Generally, each K_{ij} entry receives contributions from multiple triangles since all triangles in an triangle cluster contribute to it. Thus, in effect, the entries of K are assembled piecemeal, through cumulative contributions from integrals over individual triangles, as we walk over the mesh's triangles. The process of accumulating the integrals into K is called *assembling the system stiffness matrix*.

To be more specific, let us say the mesh has N vertices which are numbered from 0 to $N-1$. The vertices of any of the mesh's triangles may be identified in two different ways. One is through the triangle's own internal vertex numbers, as in vertex 0, vertex 1, vertex 2, enumerated in counterclockwise order. We call these the triangle's *local vertex numbers*. The other is through the vertex numbers that the triangle inherits from the mesh, as in vertex n_1, vertex n_2, vertex n_3, where $0 \le n_1, n_2, n_3 < N$. We call these the triangle's *global vertex numbers*. In assembling the system stiffness matrix we will need to determine a vertex's global index in terms of its local index. The mesh data structure of Chapter 23 is designed specifically to facilitate that determination.

25.4.3 ▪ The element stiffness matrix

Within each triangle T, the slanted faces of the pyramidal basis functions of subsection 25.4.1 appear as flat planes that rise from height 0 at one edge to height 1 at the vertex opposite that edge; see Figure A.4 on page 378 for an illustration. There are three such functions on each triangle, let's call them ϕ_1, ϕ_2, and ϕ_3, where ϕ_i is 1 at vertex i and 0 at the other two vertices. These are called the element's[93] *nodal shape functions*. The natural way to express a nodal shape function is through the element's barycentric coordinates. Following the notation of Appendix A, we have $\phi_i(\lambda_1, \lambda_2, \lambda_3) = \lambda_i$ for $i = 1, 2, 3$.

To compute an element's contributions to the system stiffness matrix K and the force vector F, we introduce the 3×3 *element stiffness matrix* k and the *element force vector* b of length 3 through

$$k_{ij} = \int_T \nabla\phi_i \cdot \nabla\phi_j \, dx, \quad b_i = \int_T f \phi_i \, dx, \tag{25.13}$$

[93] Here I have switched from talking about a *triangle* to talking about an *element*. In the FEM, an *element* is a triangle equipped with shape functions.

Listing 25.1: Assembling the system stiffness matrix K and the system force vector F.

```
initialize the matrix K and the vector F to zero
for each element e in the mesh
    compute the 3x1 element force vector b (TWB quadrature)
    compute the 3x3 element stiffness matrix k
    apply the boundary conditions
    for i in 1 2 3
        let I be the global vertex number of the element's vertex i
        for j in 1 2 3
            let J be the global vertex number of the element's vertex j
            increment K[I][J] by k[i][j]
        increment F[I] by b[i]
```

where $i, j = 1, 2, 3$. The first integral is easy to evaluate explicitly since the integrand is a constant. In fact, in Appendix A it is shown that $\nabla\phi_i \cdot \nabla\phi_j = \frac{1}{4|T|^2} \boldsymbol{e}_i \cdot \boldsymbol{e}_j$, where $\boldsymbol{e}_i$ and $\boldsymbol{e}_j$ are the element's edge vectors. It follows that

$$k_{ij} = \frac{1}{4|T|} \boldsymbol{e}_i \cdot \boldsymbol{e}_j. \tag{25.14}$$

The second integral may be evaluated by applying the TWB quadrature scheme of Chapter 24. The tricky part is to figure out where in K and F to inject the integral fragments computed in k_{ij} and b_i, that is, how to assemble the system stiffness matrix K and the system force vector F. I have given a recipe for this purpose in the from of a pseudocode in Listing 25.1. It's not hard to figure out why the recipe works if you think about it a bit, but explaining it in words will make it sound more complicated than it really is. Algorithmically, the procedure is quite straightforward. The only part that requires an explanation is the "apply the boundary conditions". I will take that up in the next subsection.

25.4.4 ▪ Applying the boundary conditions

The boundary condition $u = 0$ in (25.5) may be applied in at least two different ways that I know of, and I wouldn't be surprised if there are more variations. I will give my preferred version here. It's quite likely that you will see alternative versions in other people's codes.

Equation (25.9) gives the Galerkin approximation u_m as a linear combination of the m basis functions of the space V_m. What is m? If you track down where it first appeared, you will see that m is the number of the *interior vertices*, that is, those vertices that *do not lie on the domain's boundary*. This was necessary to ensure that functions in the space V_m vanish on the boundary, as required by the weak formulation. The unequal treatment of the interior and boundary nodes, however, is somewhat unpleasant. We avoid their segregation through the following trick.

In constructing the basis elements v_i of the space V_m, ignore the distinction between the interior and boundary vertices. Thus, construct a basis function v_i for *every vertex* in the mesh, including those on the boundary. Let's say there are m' vertices altogether. Clearly $m' > m$, and clearly it's not legitimate to apply (25.8) and (25.9) with m replaced by m'. However, it is quite easy to restore legitimacy when using the m' basis functions provided that

- in (25.10) we discard the equations corresponding to the boundary vertices; and
- in (25.9) we set the coefficients c_j of the boundary functions v_j to zero.

At the element level, the two operations above amount to the following: If the element's vertex i falls on the domain's boundary, then

- replace row i of the element stiffness matrix k by zeros;
- replace column i of the element stiffness matrix k by zeros;
- set $k_{ii} = 1$; and
- set $b_i = 0$.

This is what "apply the boundary conditions" means in Listing 25.1. You will find an extended explanation in the next chapter.

Remark 25.2. As a consequence of expanding the count of the basis functions from m to m', the system stiffness matrix K is no longer $m \times m$. It is $m' \times m'$. For the same reason, the system force vector F is of length m'.

25.5 ▪ Error analysis

It is reasonable to expect that a finer triangulation improves the quality of the solution obtained by the FEM. The central problem in the mathematical analysis of the FEM is to obtain a precise relationship between the fineness of the triangulation and the error in the FEM approximation. To make meaningful statements in that regard, the concepts of "fineness" and "error" need to be clarified.

In our program the mesh is produced through the *Triangle* library, which controls the mesh's fineness through the *maximal area* parameter a that sets an upper bound on the areas of the triangles. The default triangulation produced by *Triangle* is *quasi-uniform* in that all triangles have more or less equal areas. Furthermore, the triangles are as close to equilaterals as possible. These two properties taken together make it meaningful to talk about the "linear size" h of the triangles. Think of h as the length of a typical triangle's edge. A smaller a produces a smaller h. Since area is proportional to the square of the linear dimension, then a is roughly proportional to h^2, or equivalently, h is roughly proportional to $\sqrt{a}$.

Let u_h be the solution produced by the FEM on a grid of linear size h, and let u_{ex} be the exact solution of the boundary value problem. The error $u_{\text{ex}} - u_h$ may be measured in several different ways. Some of the most common ways are the following:

$$\text{the } \textit{energy} \text{ norm:} \qquad \|u_{\text{ex}} - u_h\|_E = \left(\int_\Omega \left| \nabla u_{\text{ex}}(x) - \nabla u_h(x) \right|^2 dx \right)^{1/2}, \tag{25.15a}$$

$$\text{the } L^2 \text{ norm:} \qquad \|u_{\text{ex}} - u_h\|_{L^2} = \left(\int_\Omega \left| u_{\text{ex}}(x) - u_h(x) \right|^2 dx \right)^{1/2}, \tag{25.15b}$$

$$\text{the } L^\infty \text{ norm:} \qquad \|u_{\text{ex}} - u_h\|_{L^\infty} = \sup_{x \in \Omega} \left| u_{\text{ex}}(x) - u_h(x) \right|. \tag{25.15c}$$

All three are meaningful in the context of n-dimensional domains. The symbol x stands for a generic point in $\boldsymbol{R}^n$. In two dimensions we write x explicitly as (x, y).

It can be shown (see, e.g., [23, pages 111 and 117], [10, page 93], and [57]) that there are constants c_1, c_2, and c_3 associated with a two-dimensional polygonal domain Ω such that when h is small, one has

$$\|u_{\text{ex}} - u_h\|_E \le c_1 h \|u_{\text{ex}}\|_{H^2}, \tag{25.16a}$$

$$\|u_{\text{ex}} - u_h\|_{L^2} \le c_2 h^2 \|u_{\text{ex}}\|_{H^2}, \tag{25.16b}$$

$$\|u_{\text{ex}} - u_h\|_{L^\infty} \le c_3 h^2 |\ln h|^{3/2} \|D^2 u_{\text{ex}}\|_{L^\infty}, \tag{25.16c}$$

Table 25.1: The errors in various norms, as computed by this chapter's FEM solver, corresponding to the *square* domain problem described on page 350 for which the exact solution is known. Figure 25.2 is a graphical representation of this table. The column with the header "nelems" lists the number of elements in each case.

a	nelems	$\|\|u_{ex} - u_h\|\|_{L^\infty}$	$\|\|u_{ex} - u_h\|\|_{L^2}$	$\|\|u_{ex} - u_h\|\|_E$
0.08	16	0.199408	0.114036	1.0555
0.04	34	0.187421	0.0873139	0.941921
0.02	79	0.0647486	0.0275582	0.521954
0.01	159	0.0526887	0.0154082	0.388679
0.005	318	0.0207487	0.00702712	0.263095
0.0025	642	0.0140025	0.00375021	0.191247
0.00125	1277	0.00524254	0.00181353	0.133232
0.00064	2427	0.00359181	0.000937034	0.0957134
0.00032	4967	0.00155731	0.000462236	0.0671078
0.00016	9946	0.000911879	0.000235085	0.0478856

where $||u_{ex}||_{H^2}$ is a Sobolev norm[94] of the solution u_{ex}, and $D^2 u_{ex}$ means the list of the second order partial derivatives of u_{ex}. In two dimensions, h is proportional to $\sqrt{a}$, as noted above; therefore the error in the L^2 norm is of order a and the error in the energy norm is of order $\sqrt{a}$. In particular, if you reduce a by a factor of $1/4$, then the error in the L^2 norm will be reduced by a factor of $1/4$, while the error in the energy norm will be reduced by a factor of $1/2$.

Table 25.1 shows the errors in various norms, as computed by this chapter's FEM solver, corresponding to the *square* domain problem described on page 350 for which the exact solution is known. I used $d = 10$ for the TWB quadrature strength in all cases. Figure 25.2 is a visual representation of that table, done in log-log plots. The dashed lines correspond to the functions $c_1\sqrt{a}$ and $c_2 a$ for some (irrelevant) choices of constants c_1 and c_2. For small values of a, the graph corresponding to $||u_{ex} - u_h||_E$ parallels $c_1\sqrt{a}$, and the graph corresponding to $||u_{ex} - u_h||_{L^2}$ parallels $c_2 a$, in agreement with the estimates (25.16).

Computing the norm $||u_{ex} - u_h||_{L^2}$ is a matter of an integration over the PDE's domain. Computing the $||u_{ex} - u_h||_E$ appears to be more complex at first sight since it involves ∇u_{ex}, which is not supplied. It turns out, however, that $||u_{ex} - u_h||_E$ may be computed without a reference to ∇u_{ex}, due to the following trick. We have

$$\begin{aligned} ||u_{ex} - u_h||_E^2 &= \int_\Omega |\nabla(u_{ex} - u_h)|^2 \, dx \\ &= \int_\Omega |\nabla u_{ex}|^2 \, dx - 2\int_\Omega \nabla u_{ex} \cdot \nabla u_h \, dx + \int_\Omega |\nabla u_h|^2 \, dx. \end{aligned}$$

The three integrals on the right-hand side may be simplified as follows. The exact solution u_{ex} satisfies the identity (25.7) for all choices of v in V. This has two consequences. First, by picking $v = u_{ex}$ we get $\int_\Omega |\nabla u_{ex}|^2 \, dx = \int_\Omega f \, u_{ex} \, dx$. Second, by picking $v = u_h$ we get

[94] You don't need to know about the Sobolev norms to appreciate the significance of these error estimates. The key part is the way h enters on their right-hand sides. Thus, the first estimate says that the error, measured in the energy norm, will decrease at least linearly in h as h goes to zero.

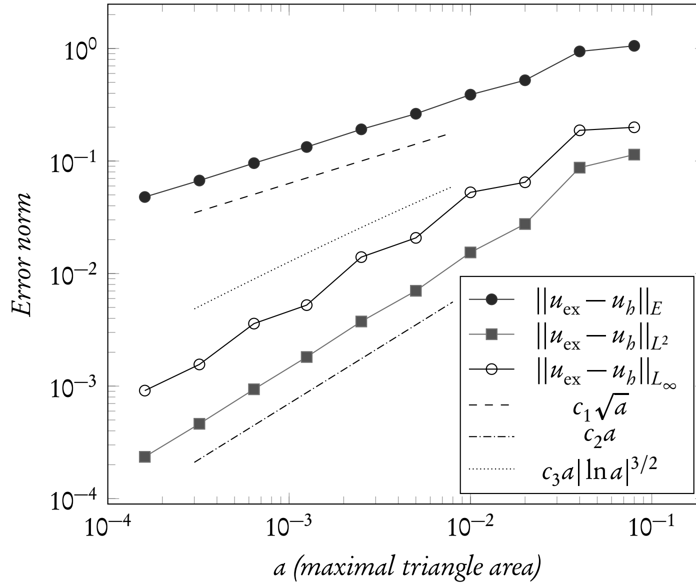


Figure 25.2: The errors in various norms, as computed by this chapter's FEM solver, corresponding to the *square* domain problem described on page 350 for which the exact solution is known. The dashed lines correspond to the functions $c_1\sqrt{a}$, $c_2 a$, and $a|\ln a|^{3/2}$ for some (irrelevant) choices of constants c_1, c_2, and c_3. For small values of a, the graph corresponding to $||u_{ex}-u_h||_E$ parallels $c_1\sqrt{a}$, and the graph corresponding to $||u_{ex}-u_h||_{L^2}$ parallels $c_2 a$, in agreement with the estimates (25.16). Table 25.1 is a tabular representation of these graphs.

$\int_\Omega \nabla u_{ex} \cdot \nabla u_h \, dx = \int_\Omega f \, u_h \, dx$. Moreover, since u_h satisfies the identity (25.8) for all v, we pick $v = u_h$ and get $\int_\Omega |\nabla u_h|^2 \, dx = \int_\Omega f \, u_h \, dx$. In view of these, the expression for the energy norm simplifies to

$$||u_{ex} - u_h||_E^2 = \int_\Omega (u_{ex} - u_h) f \, dx. \tag{25.17}$$

As you see, no gradients appear in the result; therefore computing the error's energy norm is a matter of straightforward integration.

25.6 ▪ The program

The goal of this chapter's project is to implement the FEM algorithm described in the preceding sections in order to solve the boundary value problem (25.5), or more precisely, its Galerkin approximation, (25.8). Much of the algorithm's infrastructure has been developed in the previous chapters. Specifically, we will need the files *xmalloc.[ch]* from Chapter 7 to allocate memory, the file *array.h* from Chapter 8 to construct vectors and matrices, the files *mesh.[ch]* from Chapter 23 to triangulate the domain, and the files *twb-quad.[ch]* and *plot-with-geomview.[ch]* from Chapter 24 to integrate over triangles and plot solutions. Additionally, we will need the files *problem-spec.[ch]* to describe the problem's geometry and data. The file *problem-spec.h* remains unchanged from that in *Project Triangulate* of Chapter 23; therefore a symbolic link to that file will do. The

file *problem-spec.c* is a slight variant of that file in *Project TWB Quadrature* of Chapter 24; therefore you will want a copy of, not a symbolic link to, that file. Finally, you will make links to the files *triangle.[ho]* of the *Triangle* utility, just as we did in Chapters 23 and 24. Altogether, following the suggestions in Chapters 2 and 6, the contents of this project's directory will look like this:[95]

```
$ cd fem1
$ ls -F
Makefile    plot-with-geomview.c@  problem-spec.h@  xmalloc.c@
array.h@    plot-with-geomview.h@  triangle.h@      xmalloc.h@
fem-demo.c  poisson.c              triangle.o@
mesh.c@     poisson.h              twb-quad.c@
mesh.h@     problem-spec.c         twb-quad.h@
```

The files *poisson.[ch]* contain our implementation of the FEM. The file *fem-demo.c* is a driver for demonstrating our work. I will explain their contents in the subsequent sections. The directory is named *fem1* since this is the first version of our FEM implementations. Here is the transcript of an interactive session with the program:

```
$ ./fem-demo
Usage: ./fem-demo d a
    d = the TWB quadrature strength
    a = maximal triangle area

$ ./fem-demo 10 0.002
domain is a square
nodes = 428, edges = 1217, elems = 790
system stiffness matrix is 428x428 (=183184) has 2862 nonzero entries
errors: L^infty = 0.00995973, L^2 = 0.00289579, energy norm = 0.168067
geomview output written to file square.gv

domain is a triangle with hole
nodes = 719, edges = 2013, elems = 1294
system stiffness matrix is 719x719 (=516961) has 4745 nonzero entries
geomview output written to file triangle-with-hole.gv

domain is an annulus (really a 24-gon) of radii 0.325 and 0.65
nodes = 440, edges = 1208, elems = 768
system stiffness matrix is 440x440 (=193600) has 2856 nonzero entries
geomview output written to file annulus.gv

domain is an L-shape with three holes (actually 12-gons
nodes = 443, edges = 1213, elems = 768
system stiffness matrix is 443x443 (=196249) has 2869 nonzero entries
geomview output written to file three-holes.gv
```

As you see, the program takes two arguments. The first is the desired strength of the TWB quadrature. The second is an upper bound on the areas of the triangles. The program solves a number of Poisson problems specified in the *problem-spec.c* and writes their solutions in the form of *Geomview* graphics. Figure 25.3 shows snapshots of the *Geomview* window.

[95] As in Chapter 23, the files *triangle.[ho]* link to where you keep your *Triangle* files. If you have installed *Triangle* as a library, then these are not needed; just link your program with the `-ltriangle` flag instead. See Section 23.7 for more on this.

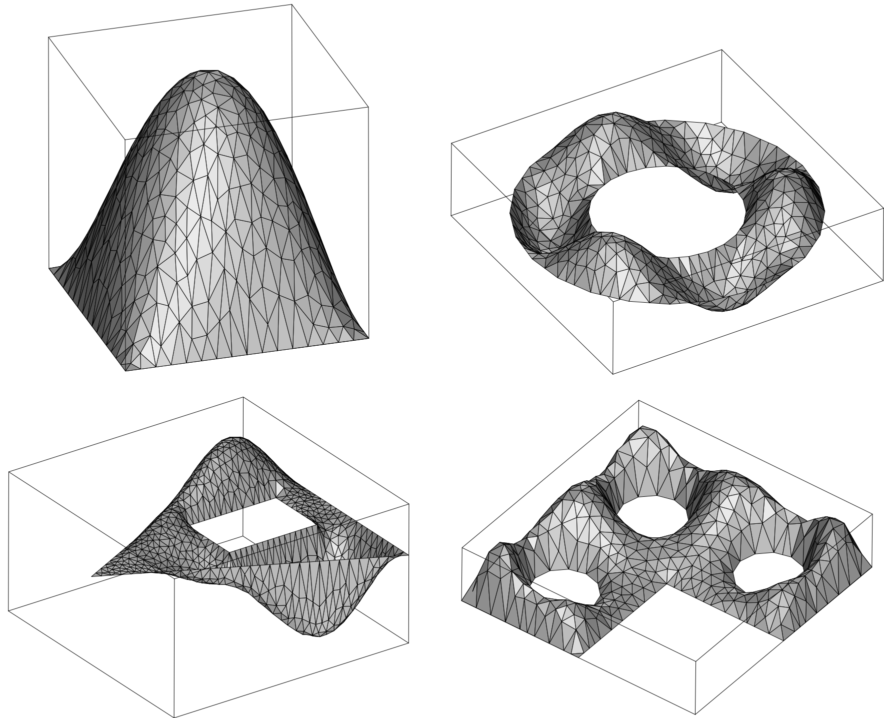


Figure 25.3: The graphs of solutions of four boundary value problems produced through the command-line `./fem-demo 10 0.002` and rendered in *Geomview*.

In the case of the square domain, we supply the program with an exact solution for comparison with the computed solution. In the transcript shown above, you will observe that in the block of output concerning the square domain, the program prints the error in the L^∞, L^2, and energy norms.

25.7 ▪ Changes in *problem-spec.c*

If you have constructed the problems specification file *problem-spec.c* from *Project TWB Quadrature* according to the instructions of Chapter 24, then it contains four domain specifications which we named *square*, *triangle with hole*, *annulus*, and *three holes*. Each domain has an associated function, named, respectively, `square_f()`, `triangle_f()`, `annulus_f()`, and `three_holes_f()`. These functions served as integrands in our TWB quadrature demo. In the current project we retain the geometry specifications but alter the four functions slightly. The new functions will serve the role of f in the Poisson equation (25.1). There is no particularly deep reason for the changes; it's just that the previous functions were more suitable for demonstrating the effectiveness of the TWB quadrature, while the new functions produce good-looking results for solutions of the boundary value problems. The file *problem-spec.h* remains unchanged. Here are my suggestions for the changes:

In the *triangle with hole* domain: Let `triangle_f()` be $30xy(2-x)(2-y)$.

Listing 25.2: The file *poisson.h* is the interface to our Poisson solver module.

```
1  #ifndef H_POISSON_H
2  #define H_POISSON_H
3  #include "problem-spec.h"
4  #include "mesh.h"
5  struct errors {
6      double Linfty;
7      double L2norm;
8      double energy;
9  };
10 void poisson_solve(struct problem_spec *spec,
11         struct mesh *mesh, int d);
12 struct errors eval_errors(struct problem_spec *spec,
13         struct mesh *mesh, int d);
14 #endif /* H_POISSON_H */
```

In the *annulus* domain: Let `annulus_f()` be what we had in Chapter 24, but change the factor 0.20 to 20.

In the *three holes* domain: Let `three_holes_f()` be $100(x^2+y^2)$.

In the *square* domain: We do things slightly differently here. Consider the function $u_{\rm ex} = 16xy(1-x)(1-y)$ on the square $0 \le x, y \le 1$. Note that $u_{\rm ex} = 0$ on the boundary. Plugging $u_{\rm ex}$ for u into the PDE in (25.2), we get $f = 32\big(x(1-x)+y(1-y)\big)$. Therefore $u_{\rm ex}$ is the exact solution of the boundary value problem corresponding to that f. This is common method of "reverse-engineering" a PDE: we plug in a function for u that satisfies the boundary conditions and find f. This provides an excellent way to test the accuracy of our FEM solver.

Thus, in the file *problem-spec.c* define the functions `square_u_exact()` and `square_f()` to correspond to the functions $u_{\rm ex}$ and f given above. Then, in the function `square()` in that file change the `u_exact` member of the `spec` structure from the previous `NULL` to `square_u_exact`.

In what follows—on line 9 in Listing 25.8, to be precise— we will set up our program so that when `u_exact` is non-`NULL`, it computes and prints the error $u_{\rm ex} - u$ in various norms, where u is the solution obtained by the FEM.

25.8 ▪ The file *poisson.h*

The file *poisson.h* is the interface to our Poisson solver module. It is shown in its entirety in Listing 25.2. It declares the prototype of the function `poisson_solve()`, which takes a problem specification, a mesh, and a number `d`, which is the desired strength of the TWB quadrature over the mesh's triangles. It applies the FEM algorithm described earlier in this chapter and ultimately evaluates the values of the coefficients c_j of the linear combination in (25.9). Since there is one c_j associated with each vertex of the mesh, the program stores the value of the c_j in the `z` member of the node structure of the node number j. These may be passed to the function `plot_with_geomview_mono()` or `plot_with_geomview_zhue()` to visualize the solution in *Geomview*. The samples in Figure 25.3 were produced that way. The headers *problem-spec.h* and *mesh.h* are

Listing 25.3: An outline of the file *poisson.c*.

```
1  ▶#include ...
2  ▶static void error_and_exit(int status, const char *file, int line) ...
3  ▶static void enforce_zero_dirichlet_bc(struct elem *ep,
4         double k[3][4]) ...
5  ▶static void compute_element_stiffness(struct elem *ep,
6         struct TWB_qdat *qdat,
7         double (*f)(double x, double y), double k[3][4]) ...
8  ▶void poisson_solve(struct problem_spec *spec,
9         struct mesh *mesh, int d) ...
10 ▶static struct errors element_errors(struct problem_spec *spec,
11        struct TWB_qdat *qdat, struct elem *ep) ...
12 ▶struct errors eval_errors(struct problem_spec *spec,
13        struct mesh *mesh, int d) ...
```

#included in *poisson.h* since the declaration of the function `poisson_solve()` refers to **struct** `problem_spec` and **struct** `mesh`.

Additionally, *poisson.h* declares a **struct** `errors` structure which is meant to hold the L^∞, L^2, and energy norms of the error $u_{\text{ex}} - u$, as described in Section 25.5. The function `eval_errors()` declared on line 12 of Listing 25.2 is called only when the user supplies an exact solution for the purpose of comparison. It computes the various error norms and returns them in an instance of **struct** `errors`.

25.9 ▪ The file *poisson.c*

The file *poisson.c* contains the implementations of the functions `poisson_solve()` and `eval_errors()` noted in the previous section and several auxiliary functions defined for internal use. Listing 25.3 gives an outline. I will describe its individual components in the following subsections. The function `error_and_exit()`, however, is identical to that in *Project* UMFPACK (see Section 12.5 on page 89); therefore I will not go over that material again. Just copy the code from *umfpack-demo2.c* to *poisson.c*.

25.9.1 ▪ The function `enforce_zero_dirichlet_bc()`

The function `enforce_zero_dirichlet_bc()` that appears on line 3 of Listing 25.3 applies the algorithm described in subsection 25.4.4 which zeros the solution on the domain's boundary. Although the algorithm is quite straightforward, I want to introduce a perhaps not-so-obvious idea here which will help with streamlining the code.

In subsection 25.4.2 we learned that the system stiffness matrix K is assembled from the 3×3 element stiffness matrices k of the mesh's individual elements. Similarly, the system force vector F is assembled from the element force vectors b, which are vectors of length 3 each. Instead of keeping track of a 3×3 matrix k and a 3-vector b as separate objects, it is more convenient to paste the two together into a 3×4 matrix whose fourth column is the vector b. I may still refer to the augmented matrix as the "element stiffness matrix" or "the matrix k", but I hope that it will be clear from the context what I mean.

Following subsection 25.4.4's instructions, if an element's ith vertex ($i = 0, 1, 2$) falls on the domain's boundary, we zero row i and column i of k and then set its (i, i) entry to 1. The following example shows the $i = 1$ case:

$$
\left[\begin{array}{ccc:c}
k_{00} & k_{01} & k_{02} & k_{03} \\
k_{10} & k_{11} & k_{12} & k_{13} \\
k_{20} & k_{21} & k_{22} & k_{23}
\end{array}\right]
\Longrightarrow
\left[\begin{array}{ccc:c}
k_{00} & 0 & k_{02} & k_{03} \\
0 & 1 & 0 & 0 \\
k_{20} & 0 & k_{22} & k_{23}
\end{array}\right]
$$

You should be able to write the function `enforce_zero_dirichlet_bc()` now. As you see in its prototype on line 3 in Listing 25.3, it receives a pointer `ep` to an element structure and the address of a 3×4 array `k`. You will set up a **`for`**-loop to go over the element's vertices, as in **`for`** `(i=0, i<3, i++)`. At vertex `i` you may access the node structure `ep→n[i]` and then examine the boundary condition there `ep→n[i]→bc`. If it equals `FEM_BC_DIRICHLET`, then the node is on the boundary, and therefore you will operate on k, as shown in the diagram above.

25.9.2 ▪ The function `compute_element_stiffness()`

The function `compute_element_stiffness()` that appears on line 5 of Listing 25.3 applies the instructions of subsection 25.4.2 to compute an element's stiffness matrix k and force vector b, which, as noted in the previous subsection, are merged into a single 3×4 matrix, also denoted by k.

The 3×3 part of the matrix k is computed according to the explicit formula (25.14). Its fourth column, which is the element's force vector b, is computed by applying a TWB quadrature to the expression for b in formula (25.13) on page 343. The function receives four data items through its arguments. These are a pointer `ep` to the element structure, a pointer `qdat` to the TWB quadrature data, a pointer `f` to the function f of the Poisson equation (25.2), and a pointer `k` to a 3×4 array where it will store the computed data.

For the most part, the function `compute_element_stiffness()` duplicates the function `integrate_over_triangle()` from Chapter 24; see Listing 24.9 on page 333. Therefore, copy the body of that function here, and modify it as follows. Line numbers refer to those in Listing 24.9.

Lines 1 through 9 remain the same. Following this, insert code to calculate the 3×3 part of the matrix k through applying the formula (25.14).

In the old code, lines 10 to the end compute the integral $\int_T f \, dx$ over the element T. These need slight adjustments since here we intend to compute not one but three integrals, $\int_T f \phi_i \, dx$, $i = 0, 1, 2$. The results are to be stored in `k[i][3]`. Since $\phi_i(\lambda_1, \lambda_2, \lambda_3) = \lambda_i$, then it suffices to replace line 17 with

```
for (i = 0; i < 3; i++)
    k[i][3] += qdat→weight * f(X,Y) * lambda[i];
```

In this connection two precautions are in order. First, since we are accumulating material in `k[i][3]`, we had better initialize it to zero. This should go somewhere before line 10. Second, line 20 multiplies the accumulated sum by the factor $|T|/|T_{\mathrm{std}}|$ and returns the result. In the modified code, each of the three `k[i][3]` needs to be multiplied by that factor. Furthermore, the return type of the new function is **`void`**, so there is nothing to return—the caller will read the computed data by examining the contents of the matrix k.

25.9.3 ▪ The function `poisson_solve()`

The function `poisson_solve()` that appears on line 8 of Listing 25.3 is one of the two externally visible functions in the file *poisson.c*. (The other is `eval_errors()`.) I have shown it in its entirety in Listings 25.4 and 25.5. The function is a somewhat long (perhaps too long) because it performs several loosely related tasks which ultimately produce the FEM approximation of the solution of the boundary value problem (25.5). It may be possible to split the function into simpler components, but I don't see a neat way for doing that. You may. See what you can do.

The function is invoked with three arguments. These are a pointer `spec` to a problem specification structure, a pointer `mesh` to a precomputed mesh structure, and a number `d` which is the desired TWB quadrature strength for integrating over triangles. It assembles the system stiffness matrix K and the system force vector F, calls UMFPACK to solve the linear system of equations (25.12), and then places the coefficient c_j in the `z` member of the node structure of the node number j for each j.

A typical system stiffness matrix K is huge and very sparse. We have no intention of wasting memory in storing K in its entirety. Rather, we store only its nonzero entries in the compressed column storage (CCS) form as seen in Chapters 11 and 12. More precisely, we store K's nonzero entries in UMFPACK's *triplet form* as explained in Section 12.6 (page 90) and then convert from the triplet form to the CCS form by applying UMFPACK's `umfpack_di_triplet_to_col()` function. Let's see how this is done by examining Listing 25.4:

Line 10. We are going to need the TWB quadrature data for integration over the triangles, so we might as well retrieve it right now. Following the remark in Example 24.3 on page 324, we set the second argument of `twb_qdat()` to `NULL` since we have no need for the number of quadrature points in the rest of the code. If you *do* want to know the number of quadrature points, then follow Example 24.2 on page 324.

Lines 12–14. The major task of the function `poisson_solve()` is to implement the idea shown in the pseudocode in Listing 25.1 on page 344. There, we go over the mesh's triangles, and in each triangle we compute the element stiffness matrix `k` and then in effect "inject" `k` into the global stiffness matrix `K`, that is to say, increment the values of `K[I][J]` by `k[i][j]` for `i,j=0,1,2`. We have no intention, however, of allocating memory to store the massively sparse matrix `K`. We merely store the incrementation data, that is, the indices `I` and `J`, and the increment amount `k[i][j]` into UMFPACK's triplet vectors `Ti`, `Tj`, and `Tx`; see Section 12.6, page 90.

That is all to come later. What we need to do right now is to determine suitable lengths for the triplet vectors. They should be long enough to hold all the incrementation data that is handed to them. How may increments are there? The element stiffness matrix is 3×3. The mesh has `mesh→nelems` elements. Therefore, there will be a total of `3*3*mesh→nelems` increments. Ergo, the triplet vectors should be that long. That's what we see on lines 12–14.

Lines 16–18. UMFPACK's triplet vectors are there for the programmer's convenience. What we really need are the corresponding CCS vectors `Ap`, `Ai`, and `Ax`. As we saw in Section 12.6, UMFPACK's `umfpack_di_triplet_to_col()` converts the triplet format to the CCS format. In lines 16–18 we allocate memory for the CCS vectors.

Listing 25.4: The top half of the function `poisson_solve()` in the file *poisson.c*.

```
void poisson_solve(struct problem_spec *spec, struct mesh *mesh, int d)
{
    double k[3][4];
    int *Ti, *Tj, *Ai, *Ap;
    double *Tx, *Ax, *F, *U;
    int status;
    void *Symbolic = NULL;
    void *Numeric = NULL;
    int i, j, r, s;
    struct TWB_qdat *qdat = twb_qdat(&d, NULL);

    make_vector(Ti, 3*3*mesh->nelems);
    make_vector(Tj, 3*3*mesh->nelems);
    make_vector(Tx, 3*3*mesh->nelems);

    make_vector(Ap, 1 + mesh->nnodes);
    make_vector(Ai, 3*3*mesh->nelems);
    make_vector(Ax, 3*3*mesh->nelems);

    make_vector(F, mesh->nnodes);
    make_vector(U, mesh->nnodes);

    for (i = 0; i < mesh->nnodes; i++)
        F[i] = 0.0;

    s = 0;
    for (r = 0; r < mesh->nelems; r++) {
        struct elem *ep = &mesh->elems[r];
        compute_element_stiffness(ep, qdat, spec->f, k);
        enforce_zero_dirichlet_bc(ep, k);
        for (i = 0; i < 3; i++) {
            int I = ep->n[i]->nodeno;
            for (j = 0; j < 3; j++) {
                if (k[i][j] != 0.0) {
                    int J = ep->n[j]->nodeno;
                    Ti[s] = I;
                    Tj[s] = J;
                    Tx[s] = k[i][j];
                    s++;
                }
            }
            F[I] += k[i][3];
        }
    }
    // continued in Listing 25.5
```

Listing 25.5: The bottom half of the function `poisson_solve()` in the file *poisson.c*.

```
46      // continued from Listing 25.4
47      status = umfpack_di_triplet_to_col(
48              mesh→nnodes, mesh→nnodes, s,
49              Ti, Tj, Tx, Ap, Ai, Ax, NULL);
50      if (status ≠ UMFPACK_OK)
51          error_and_exit(status, __FILE__, __LINE__);
52
53      printf("system stiffness matrix is %dx%d (=%d) "
54              "has %d nonzero entries\n",
55              mesh→nnodes, mesh→nnodes,
56              mesh→nnodes * mesh→nnodes,
57              Ap[mesh→nnodes]);
58
59      // symbolic analysis
60      status = umfpack_di_symbolic(
61              mesh→nnodes, mesh→nnodes,
62              Ap, Ai, Ax, &Symbolic, NULL, NULL);
63      if (status ≠ UMFPACK_OK)
64          error_and_exit(status, __FILE__, __LINE__);
65
66      // numeric analysis
67      status = umfpack_di_numeric(
68              Ap, Ai, Ax, Symbolic, &Numeric, NULL, NULL);
69      if (status ≠ UMFPACK_OK)
70          error_and_exit(status, __FILE__, __LINE__);
71
72      // solve system
73      status = umfpack_di_solve(UMFPACK_A,
74              Ap, Ai, Ax, U, F, Numeric, NULL, NULL);
75      if (status ≠ UMFPACK_OK)
76          error_and_exit(status, __FILE__, __LINE__);
77
78      for (i = 0; i < mesh→nnodes; i++)
79          mesh→nodes[i].z = U[i];
80
81      free_vector(Ti);
82      free_vector(Tj);
83      free_vector(Tx);
84      free_vector(Ap);
85      free_vector(Ai);
86      free_vector(Ax);
87      free_vector(F);
88      free_vector(U);
89      umfpack_di_free_symbolic(&Symbolic);
90      umfpack_di_free_numeric(&Numeric);
91  }
```

According to the CCS specifications (see Chapter 11), if the matrix is $n \times n$, then the length of the vector Ap should be $n+1$. The n in the case of the system stiffness matrix is mesh→nnodes. That explains the choice of the size of Ap on line 16.

Again, according to the CCS specifications, the lengths of the vectors Ai and Ax should be equal to the number of nonzero entries, nz, of the matrix K. Unfortunately the value of nz is not known ahead of the time; therefore we cannot allocate vectors of precise length here. However, we know that nz cannot be more than the lengths of the triplet vectors, that is, 3*3*mesh→nelems, since that's the number of items injected into the matrix K. We go with this overestimate and allocate memory for the vectors Ai and Ax accordingly.

Lines 20–21. We allocate memory for the system force vector F and the solution vector U of (25.12). I have changed the notation from c to U merely for aesthetic reasons. You may change it back to c if you want.

Line 23. The system force vector F is going to be constructed by accumulating the element force vectors. Here we initialize F to zeros in preparation for that process.

Lines 26–44. This set of lines implements the pseudocode in Listing 25.1 on page 344. The variable s serves as an index into the triplet vectors Ti, Tj, and Tx. We initialize it to zero and then increment it with every injection into the system stiffness matrix.

The **for**-loop on line 27 walks over the mesh's triangles. The rth element is mesh→elems[r]. On line 28 we introduce the notation ep as a shorthand for the address of that element. (Think of ep as an "element pointer".) In the subsequent two lines we compute ep's 3×4 stiffness matrix k and then apply the zero Dirichlet conditions, as discussed earlier. The rest of the block injects the contents of the element stiffness matrix into the system stiffness matrix. Refer to the pseudocode in Listing 25.1 for explanation.

Line 34 skips over those k[i][j] that are zero. This is not absolutely essential since adding zero to anything is immaterial. In that respect, the solution is unaffected whether we skip over k[i][j] or not. Inserting a zero, however, *does affect* the *sparseness* of the system stiffness matrix. Any entry inserted here, whether zero or not, becomes a part of the sparse matrix which is eventually handed to UMFPACK. In skipping the zeros we are making the stiffness matrix sparser, hence increasing UMFPACK's efficiency.

Continuing into Listing 25.5:

Lines 47–76. This is a routine application of UMFPACK along the lines described in Sections 12.5 and 12.6. There is nothing new here. If everything proceeds as expected, the solution of the system of equations (25.12) will be computed and placed in the vector U.

Lines 78–90. We copy the entries of the solution vector U into the z member of the node structure of the corresponding vertices. We free the previously allocated memory, and we are done.

25.9.4 ▪ The function element_errors()

The function element_errors() that appears on line 10 of Listing 25.3 on page 351 measures the error $u_{\text{ex}} - u$ on a given element, say T, between the user-supplied exact

solution u_{ex} and the FEM's solution u. Specifically, it computes the values

$$\max_T |u_{\text{ex}} - u|, \qquad \int_T |u_{\text{ex}} - u|^2 \, dx, \qquad \int_T (u_{\text{ex}} - u) f \, dx, \tag{25.18}$$

which are fragments of the norms defined in equations (25.15).[96] The fragments are put together in function `eval_errors()`, which is the subject of the next subsection.

To evaluate the second and third expressions in (25.18), we apply the TWB quadrature in the usual way. The barycentric coordinates `lambda[]` and the quadrature points `X` and `Y` are calculated as in all other TWB quadrature calculations. The value of the function u_{ex} at a quadrature point is `spec→u_exact(X,Y)`. A new feature is the need for the value u of the FEM's solution at the quadrature points. It is given by

$$\sum_{i=1}^{3} \lambda_i z_i,$$

where z_i are the values of u at the triangle's vertices, whose values are available in `ep→n[i]→z`.

To evaluate the first expression in (25.18), we calculate the maximum of the differences $|u_{\text{ex}} - u|$ at the element's TWB quadrature points. Admittedly, this is not truly the maximum over the whole triangle since the maximum may occur at a place other than a quadrature point; however, for smallish elements and moderate quadrature strength, it should come close. A possible improvement would be to include the triangle's vertices among the points where the differences are evaluated. That will require a trivial addition to the function. Do it if you feel so inclined.

We pack the values of the three expression (25.18) into a **struct** `errors` and return that structure to the caller. Therefore the function's body will include, in part,

```
struct errors elem_errors;
elem_errors.Linfty = elem_errors.L2norm = elem_errors.energy = 0.0;
... compute elem_errors.Linfty, etc. ...
return elem_errors;
```

I will let you implement the function `element_errors()`. Let me close with a caution regarding the evaluation of $|u_{\text{ex}} - u|$. There are at least two "absolute value" functions in C. The function `abs()` computes the absolute values of integers. The function `fabs()` computes the absolute values of floating point numbers. Be careful about which one you use. They are not interchangeable!

25.9.5 ▪ The function `eval_errors()`

The function `eval_errors()` that appears on line 12 of Listing 25.3 on page 351 is called only when the user supplies an exact solution, u_{ex}. It computes the three error norms $||u_{\text{ex}} - u||_{L^\infty}$, $||u_{\text{ex}} - u||_{L^2}$, and $||u_{\text{ex}} - u||_E$ (see (25.15)), where u is the FEM's own solution. These may be used to analyze, verify, and debug the correctness of the solver. Listing 25.6 shows the function in its entirety. Let us look at its details.

Line 4. This structure will hold the values of the three error norms noted above. A copy of the structure will be returned to the caller.

[96] The third expression is related to the energy norm through the formula (25.17).

Listing 25.6: The function `eval_errors` in the file *poisson.c*.

```
1  struct errors eval_errors(struct problem_spec *spec,
2          struct mesh *mesh, int d)
3  {
4      struct errors errors;
5      struct TWB_qdat *qdat = twb_qdat(&d, NULL);
6      errors.Linfty = errors.L2norm = errors.energy = 0.0;
7      for (int i = 0; i < mesh->nelems; i++) {
8          struct elem *ep = &mesh->elems[i];
9          struct errors elem_errors = element_errors(spec, qdat, ep);
10         errors.L2norm += elem_errors.L2norm;
11         errors.energy += elem_errors.energy;
12         if (elem_errors.Linfty > errors.Linfty)
13             errors.Linfty = elem_errors.Linfty;
14     }
15     errors.L2norm = sqrt(errors.L2norm);
16     errors.energy = sqrt(errors.energy);
17     return errors;
18 }
```

Listing 25.7: An outline of the file *fem-demo.c*.

```
1  ▶ #include ...
2  ▶ static void do_demo(struct problem_spec *spec,
3          double a, int d, char *gv_filename) ...
4  ▶ static void show_usage(char *progname) ...
5  ▶ int main(int argc, char **argv) ...
```

Line 6. The error norms are going to be computed one element at a time. We initialize the norms to zero in preparation for the accumulation process.

Line 9. The `elem_errors` is yet another instance of a **`struct`** `errors`. It captures the element-level errors returned in a call to the function `element_errors()`. Following that call, we accumulate the returned values in `errors.L2norm` and `errors.energy`. We also adjust `errors.Linfty` if the element's error value exceeds the previous values.

Lines 15 and 16. We take square roots in accordance with the definitions (25.15).

25.10 ▪ The file *fem-demo.c*

The file *fem-demo.c* is a driver for demonstrating our Poisson solver. Listing 25.7 shows an outline. The functions `show_usage()` and `main()` that appear on lines 4 and 5 of that listing are essentially identical to those in *Project TWB quadrature*. The function `main()` expects to read two numbers, `d` and `a`, from its command-line. The first number is the strength of the desired TWB quadrature for integrating over the elements. The second number is an upper bound on the areas of the mesh's triangles. Thus we expect `argc` to be 3. If it's not, we call `show_usage()` to print a brief usage message to `stderr` and then exit. This is exactly what we did in *Project TWB quadrature*, so there is no need to elaborate.

Listing 25.8: The function do_demo() in the file *fem-demo.c*.

```
1  static void do_demo(struct problem_spec *spec,
2          double a, int d, char *gv_filename)
3  {
4      struct mesh *mesh;
5      mesh = make_mesh(spec, a);
6      printf("nodes = %d, edges = %d, elems = %d\n",
7              mesh→nnodes, mesh→nedges, mesh→nelems);
8      poisson_solve(spec, mesh, d);
9      if (spec→u_exact ≠ NULL) {
10         struct errors errors = eval_errors(spec, mesh, d);
11         printf("errors: L^infty = %g, L^2 = %g, energy norm = %g\n",
12             errors.Linfty, errors.L2norm, errors.energy);
13     }
14     plot_with_geomview_zhue(mesh, gv_filename);
15     free_mesh(mesh);
16 }
```

The function do_demo() that appears on line 2 of Listing 25.7 runs the demo for a single problem. It receives the problem specification in the spec argument, the values of maximal triangle area a, TWB quadrature strength d, and a file name for writing a *Geomview* script for a three-dimensional rendering of the solution. Listing 25.8 shows the implementation of do_demo(). It is mostly self-explanatory, so there is no need to elaborate other than pointing out that on line 9 we check the value of spec→u_exact. If it is not NULL, then we surmise that the user has supplied an *exact solution*, and therefore we call eval_errors() to compute the error norms and print them to stdout.

25.11 ▪ Further reading

For the requisite theory of PDEs you may begin with the books by Renardy and Rogers [55], Mattheij, Rienstra, and ten Thije Boonkkamp [43], or Ladyzhenskaya [39], where you will find accessible introductions to the subject. The books by Quarteroni [54] and Steinbach [64] provide quick overviews of the analysis of PDEs and relate them to numerical computations and the FEM. None of these makes for a "light reading", however; they require a moderate knowledge of mathematical analysis at an advanced undergraduate or early graduate level. I am afraid there is no easier way.

You will find more detailed (but significantly more advanced) accounts of PDEs in Evans [19], Gilbarg and Trudinger [22], and (the classic) Lions and Magenes [41]. For the general theory of Sobolev spaces, a specialist may consult Adams and Fournier [1] and Rudin [56]. To delve seriously into the FEM, a good understanding of the Sobolev spaces is indispensable.

All of the above focus on the regularity (that is, the differentiability properties) of the functions while assuming that the domain's boundary $\partial\Omega$ is "sufficiently smooth". Kozlov, Maz'ya, and Rossmann [37] and Maz'ya and Rossmann [44] introduce a theory of *weighted Sobolev spaces* that extends the coverage to domains with lesser smoothness requirements. The articles of Soane and Rostamian [63] and Soane, Suri, and Rostamian [62] develop theory and finite element analysis in this setting.

As to books dealing with the analysis and implementation of the FEM, you will find excellent introductions in Gockenbach [23], Braess [10], Szabó and Babuška [72], Brenner

and Scott [11], and Shapira [59]. The classic books by Ciarlet [12] and Axelsson and Barker [5] provide almost encyclopedic coverage of the analytic aspects of the FEM.

25.12 ▪ *Project FEM 1*

Complete the program, compile, and test. Since we have an exact solution in the case of a square domain, tabulate the error as a function of the maximal area of the triangles.

Chapter 26

Finite elements: Nonzero boundary data

Prerequisites: Chapters 7, 8, 11, 12, 22, 23, 24, 25, and Appendix A

26.1 ▪ The problem

In this chapter we generalize the previous chapter's program to solve the boundary value problem

$$\nabla\cdot(\eta\nabla u)+f=0 \qquad \text{in } \Omega, \tag{26.1a}$$

$$u=g \qquad \text{on } \Gamma_D, \tag{26.1b}$$

$$(\eta\nabla u)\cdot\boldsymbol{n}=h \qquad \text{on } \Gamma_N, \tag{26.1c}$$

where Ω is a polygonal domain in $\boldsymbol{R}^2$, and Γ_D and Γ_N are two complementary subsets of Ω's boundary $\partial\Omega$, that is, $\Gamma_D\cap\Gamma_N=\emptyset$ and $\Gamma_D\cup\Gamma_N=\partial\Omega$. The functions g and h are called the problem's *Dirichlet* and *Neumann* boundary data, respectively. Thus, we refer to Γ_D and Γ_N as the *Dirichlet and Neumann parts of the boundary*. The $\boldsymbol{n}$ in (26.1c) is the outward unit normal vector to the domain's boundary. The vector $-\eta\nabla u$ is called the *flux vector*. Figure 26.1 shows a sample domain Ω and the associated boundary parts. Although the domain depicted there has rounded boundary curves, our program assumes that the boundaries consist of piecewise-straight line segments.

In the previous chapter the part Γ_D was the entire boundary, and Γ_N was the empty set. The other extreme, where Γ_N is the entire boundary and Γ_D is the empty set, is also of some interest, but it calls for a certain (not too difficult) special handling which I will avoid in order to keep the exposition as simple as possible. Therefore, from now on I will assume that Γ_D *is not the empty set*. (To be more precise, I assume that Γ_D has a positive length.) The case where Γ_D is the entire boundary, on the other hand, is not special and falls within the scope of this chapter.

The coefficient η is not necessarily a constant; it may be any (measurable) function that takes values in some bounded interval $[\alpha,\beta]$, where $\alpha>0$, that is,

$$0<\alpha\le\eta(x,y)\le\beta \quad \text{for all } (x,y)\in\Omega.$$

In particular, when $\eta(x,y)\equiv 1$, (26.1a) reduces to the Poisson's equation $\nabla\cdot\nabla u+f=0$ of Chapter 25.

The objective of this chapter is to extend the previous chapter's FEM code to solve the boundary value problem (26.1). Specifically, given a polygonal domain Ω in $\boldsymbol{R}^2$, the

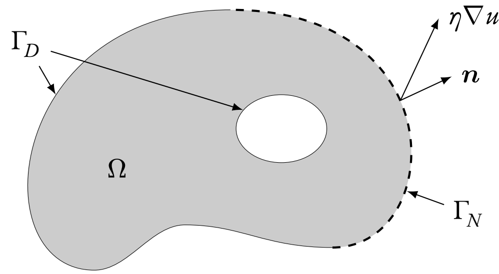


Figure 26.1: A sample two-dimensional domain Ω with a hole. The boundary part Γ_D, drawn in solid lines, consists of two curves. The boundary part Γ_N is drawn in a dashed line. The outward unit normal $\boldsymbol{n}$ and the flux vector $\eta\nabla u$ are shown at an arbitrary point on Γ_N. Our FEM implementation assumes that the domain is polygonal; therefore you should imagine the boundary curves as piecewise-straight line segments.

boundary parts Γ_D and Γ_N, and the functions η, f, g, and h, the program will find an arbitrarily close approximation to the function u that satisfies the three equations in (26.1).

Remark 26.1. The Neumann boundary condition (26.1c) is equivalent to $\eta \partial u/\partial n = h$, that is, $\partial u/\partial n = h/\eta$, where $\partial u/\partial n$ is the directional derivative of u in the direction of the outward normal $\boldsymbol{n}$. It is likely that you will see the Neumann boundary condition expressed in this form in some books and articles.

Remark 26.2. If the boundary value problem models a diffusion process in a material that occupies the domain Ω, then the coefficient η is the material's *diffusivity*. In the case of heat conduction, η is the material's *conductivity*. In general, the physical meaning of η depends on the application. If η is constant throughout Ω, then the material is said to be *homogeneous*; otherwise it is *inhomogeneous*.

Remark 26.3. This chapter's theory and implementation may be generalized in quite a straightforward way to the equation

$$\nabla \cdot (A\nabla u) + f = 0, \tag{26.2}$$

where $A = \big[A_{ij}(x,y)\big]$ is a 2×2 matrix at any $(x,y)\in\Omega$. Equation (26.1a) is a special case of (26.2) where $A(x,y) = \eta(x,y)I$, where I is the 2×2 identity matrix. If $A(x,y)$ is a multiple of identity, the material is said to be *isotropic* at (x,y); otherwise it is *anisotropic*.

26.2 ▪ The weak formulation

In the weak formulation of the boundary value problem in Chapter 25, I steered away from technical jargon and couched the statements in a somewhat vague and "soft" terminology. In this chapter I will take the opposite approach: I will present the weak formulation properly in terms of Sobolev spaces. Even if you are not familiar with the theory of Sobolev spaces, you should be able to read through this material and get the gist of it to an extent that enables you to proceed with the implementation of the project.

In what follows, $H^1(\Omega)$ is the Sobolev space of functions defined on Ω which are square integrable and whose first order generalized derivatives are also square integrable. Let Γ_D and Γ_N be the boundary parts introduced in the previous section. We write $H^1_{g,\Gamma_D}(\Omega)$ for

the subset of functions in $H^1(\Omega)$ that take on (in the sense of traces) the value g on the Γ_D part of the boundary and $H^1_{0,\Gamma_D}(\Omega)$ for the subset of functions in $H^1(\Omega)$ that vanish (i.e., take on the zero value) on Γ_D. Let us note that $H^1_{g,\Gamma_D}(\Omega)$ is an *affine space* in $H^1(\Omega)$ if g is nonzero. On the other hand, $H^1_{0,\Gamma_D}(\Omega)$ is a proper subspace of $H^1(\Omega)$.

Let $u \in H^1_{g,\Gamma_D}(\Omega)$ be the solution of the boundary value problem (26.1), and let $v \in H^1_{0,\Gamma_D}(\Omega)$ be an arbitrary function. Multiply (26.1a) by v and integrate over Ω,

$$\int_\Omega v \operatorname{div}(\eta \nabla u)\, dx + \int_\Omega f v\, dx = 0;$$

then apply Green's identity ((25.4) on page 339) to the first integral to get

$$\int_{\partial\Omega} v \eta \nabla u \cdot \boldsymbol{n}\, da - \int_\Omega \eta \nabla u \cdot \nabla v\, dx + \int_\Omega f v\, dx = 0.$$

Split the boundary integral into the sum of integrals over Γ_D and Γ_N. The former is zero since $v = 0$ on Γ_D. The latter may be simplified by applying (26.1c). After rearranging the terms we get the identity

$$\int_\Omega \eta \nabla u \cdot \nabla v\, dx = \int_\Omega f v\, dx + \int_{\Gamma_N} h v\, da,$$

which holds for all v in $H^1_{0,\Gamma_D}(\Omega)$. Conversely, if the above holds for a $u \in H^1_{g,\Gamma_D}(\Omega)$ and all v in $H^1_{0,\Gamma_D}(\Omega)$, one may deduce that u is a solution of the original boundary value problem. (The derivatives should be interpreted as *generalized derivatives*, but that technical issue is beyond the scope of this book.) This leads to the following *weak formulation* of the boundary value problem:

Find $u \in H^1_{g,\Gamma_D}(\Omega)$ such that

$$\int_\Omega \eta \nabla u \cdot \nabla v\, dx = \int_\Omega f v\, dx + \int_{\Gamma_N} h v\, da \quad \text{for all } v \in H^1_{0,\Gamma_D}(\Omega). \tag{26.3}$$

Remark 26.4. In the special case when h is identically zero on Γ_N, the weak formulation (26.3) reduces to the following:

Find $u \in H^1_{g,\Gamma_D}(\Omega)$ such that

$$\int_\Omega \eta \nabla u \cdot \nabla v\, dx = \int_\Omega f v\, dx \quad \text{for all } v \in H^1_{0,\Gamma_D}(\Omega).$$

You may note that there is no explicit reference to a Neumann boundary condition here. In a sense, omitting h in the weak formulation is equivalent to taking h as identically zero. Because of this, a zero Neumann boundary condition is also known as a *natural boundary condition*.

26.3 ▪ The Galerkin approximation

Following the ideas of Chapter 25, we triangulate Ω with a quasi-uniform mesh whose fineness is determined by a parameter a that specifies an upper bound on triangle areas. The number m of the mesh's nodes will increase as a decreases. Recall the construction, in Section 25.4, of the function space V_m whose basis consisted of the pyramidal functions depicted in Figure 25.1 on page 342. The preliminary construction in that section placed one pyramid at each *interior* node of the mesh. Boundary nodes were excluded since we were dealing with zero Dirichlet data. In the present chapter, the space V_m is constructed in the same way; however, there is a pyramidal basis function at *every* node, including the boundary nodes, since we are dealing with general boundary conditions. Of course, those parts of the pyramids that lie outside the domain Ω are immaterial to our purposes. It may be shown (see, e.g., [12]) that V_m is a linear subspace of $H^1(\Omega)$, and that a function in $H^1(\Omega)$ may be approximated with a function in V_m with any desired accuracy if m is sufficiently large.

Let us write $\mathscr{I} = \{1,2,\dots,m\}$ for the index set of the triangulation's nodes, and let us split $\mathscr{I}$ into the disjoint union $\mathscr{I} = \mathring{\mathscr{I}} \cup \mathscr{I}_D$, where $\mathscr{I}_D$ is the subset of $\mathscr{I}$ corresponding to the vertices that fall on the part Γ_D of the domain's boundary, and $\mathring{\mathscr{I}}$ is the rest of $\mathscr{I}$. We will write g_i for the values of the Dirichlet data g at the point v_i when $i \in \mathscr{I}_D$.

Here we introduce the finite-dimensional counterparts of the spaces $H^1_{0,\Gamma_D}(\Omega)$ and $H^1_{g,\Gamma_D}(\Omega)$ that enter the weak formulation (26.3):

$$V_{0,m} = \Big\{ \sum_{i\in\mathring{\mathscr{I}}} c_i v_i : \text{ all possible choices of coefficients } c_i \Big\} = \operatorname{span}\{v_i : i \in \mathring{\mathscr{I}}\},$$

$$V_{g,m} = \Big\{ \sum_{i\in\mathring{\mathscr{I}}} c_i v_i + \sum_{i\in\mathscr{I}_D} g_i v_i : \text{ all possible choices of coefficients } c_i \Big\}.$$

Then the Galerkin approximation of the weak formulation (26.3) takes the following form:

> Find $u_m \in V_{g,m}$ such that
>
> $$\int_\Omega \eta \nabla u_m \cdot \nabla v \, dx = \int_\Omega f v \, dx + \int_{\Gamma_N} h v \, da \quad \text{for all } v \in V_{0,m},$$

or equivalently, the following:

> Find $\{c_j\}_{j\in\mathring{\mathscr{I}}}$ such that
>
> $$\int_\Omega \eta \nabla\Big(\sum_{j\in\mathring{\mathscr{I}}} c_j v_j + \sum_{j\in\mathscr{I}_D} g_j v_j \Big) \cdot \nabla v_i \, dx = \int_\Omega f v_i \, dx + \int_{\Gamma_N} h v_i \, da \quad \text{for all } i \in \mathring{\mathscr{I}}. \tag{26.4}$$

The formulation (26.4) is a linear system of $\mathring{m}$ equations in the $\mathring{m}$ unknowns c_j, where $\mathring{m}$ is the cardinality of the set $\mathring{\mathscr{I}}$. In analogy with (25.12) (page 341) let us write this as $\mathring{K}c = \mathring{b}$. The system may be assembled and solved, in principle, along the lines developed for the simpler case of Chapter 25. In practice, however, the following slight reformulation of (26.4) will help to streamline the program's algorithm. The purpose of

the reformulation is to remove references to the index sets $\mathring{\mathscr{I}}$ and $\mathscr{I}_D$ in (26.4) in favor of the larger but simpler index set $\mathscr{I}$. Toward that end, we express

$$u_m = \sum_{j \in \mathring{\mathscr{I}}} c_j v_j + \sum_{j \in \mathscr{I}_D} g_j v_j = \sum_{j \in \mathscr{I}} c_j v_j, \tag{26.5}$$

thus extending the count of the unknowns c_i from $\mathring{m}$ to m. At the same time, we append $m - \mathring{m}$ equations of the type $c_j = g_j$, $j \in \mathscr{I}_D$, to the previous system, thus arriving at a system $Kc = b$ of m equations in m unknowns with the following general structure:

$$\begin{pmatrix} k_{11} & k_{12} & \cdots & k_{1j} & \cdots & k_{1m} \\ k_{21} & k_{22} & \cdots & k_{2j} & \cdots & k_{2m} \\ \vdots & & & & & \\ 0 & 0 & \cdots & 1 & \cdots & 0 \\ \vdots & & & & & \\ k_{m1} & k_{m2} & \cdots & k_{mj} & \cdots & k_{mm} \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_j \\ \vdots \\ c_m \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ g_j \\ \vdots \\ b_m \end{pmatrix}. \tag{26.6}$$

Row j expresses the constraint $c_j = g_j$, assuming that the index j is in the set $\mathscr{I}_D$. There is one such row for every vertex that falls on Γ_D. We may exploit that row's simple structure to eliminate the rest of the entries in column j, as in

$$\begin{pmatrix} k_{11} & k_{12} & \cdots & 0 & \cdots & k_{1m} \\ k_{21} & k_{22} & \cdots & 0 & \cdots & k_{2m} \\ \vdots & & & & & \\ 0 & 0 & \cdots & 1 & \cdots & 0 \\ \vdots & & & & & \\ k_{m1} & k_{m2} & \cdots & 0 & \cdots & k_{mm} \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_j \\ \vdots \\ c_m \end{pmatrix} = \begin{pmatrix} b_1 - g_j k_{1j} \\ b_2 - g_j k_{2j} \\ \vdots \\ g_j \\ \vdots \\ b_m - g_j k_{mj} \end{pmatrix}. \tag{26.7}$$

The elimination that takes (26.6) to (26.7) is by no means essential, but it helps to make the sparse coefficient matrix K more sparse and thus increase UMFPACK's efficiency.

In view of the weak formulation in (26.4) and the compact form in (26.5) of u_m, the entries K_{ij} of the *system stiffness matrix* K are given by

$$K_{ij} = \int_\Omega \eta \nabla v_i \cdot \nabla v_j \, dx \, dy.$$

As we saw in Chapter 25, however, there is little use for such a "node-based" formula. The "element-based" approach is easier to implement. Thus, we introduce the 3×3 *element stiffness matrix* k on the triangle T, given by

$$k_{ij} = \int_T \eta \nabla \phi_i \cdot \nabla \phi_j \, dx \, dy, \quad i, j = 1, 2, 3,$$

where $\phi_i(x, y) = \lambda_i$ is the nodal shape function corresponding to the vertex i of the triangle T. Here $(\lambda_1, \lambda_2, \lambda_3)$ are the barycentric coordinates of the point (x, y).

Since a shape function is a first degree polynomial, then $\nabla \phi_i$ is a constant vector, and therefore it may be pulled out of the integration sign. Applying (A.10) (on page 379), the expression for k_{ij} reduces to

$$k_{ij} = \left(\frac{1}{4|T|^2} \boldsymbol{e}_i \cdot \boldsymbol{e}_j \right) \left(\int_T \eta \, dx \, dy \right). \tag{26.8}$$

In particular, when $\eta \equiv 1$, the integral equals the area $|T|$ of the triangle; therefore (26.8) reduces to Chapter 25's (25.14).

Remark 26.5. Rearranging the expression (26.8) into

$$k_{ij} = \left(\frac{1}{4|T|} e_i \cdot e_j \right) \left(\frac{1}{|T|} \int_T \eta \, dx \, dy \right) \tag{26.9}$$

we see that the element stiffness matrix is exactly the previous chapter's element stiffness matrix multiplied by the average of η over the element. This observation should help you with adapting your previous code to the current situation.

26.4 ▪ The program

The implementation of the FEM solver for the boundary value problem (26.1) amounts to making small changes to the solver of Chapter 25. I assume that you have read that chapter and implemented its program. In this chapter I will focus only on the differences. Here is what my directory looks like:

```
$ cd fem2
$ ls -F
Makefile        mesh.h@                  problem-spec.h@
array.h@        plot-with-geomview.c@   twb-quad.c@
fem-demo.c      plot-with-geomview.h@   twb-quad.h@
gauss-quad.c@   poisson.c                xmalloc.c@
gauss-quad.h@   poisson.h                xmalloc.h@
mesh.c@         problem-spec.c
```

The file *problem-spec.h* remains unchanged from that in *Project Triangulate* of Chapter 23; therefore a symbolic link to that file will do. The file *problem-spec.c* is substantially different; make a copy from Chapter 25, and edit. In *problem-spec.c* we define four problems, called `square1()`, `square2()`, `square3()`, `square4()`, each corresponding to a boundary value problem on the unit square $\Omega = (0,1) \times (0,1)$. Figures 26.2 and 26.3 provide all the necessary details. Problems `square1()` and `square2()` test your code's ability to apply nonzero Dirichlet and Neumann boundary data, respectively. They share the same forcing function f and exact solution u_{ex}. Problem `square3()` tests your code's ability to handle inhomogeneous media, that is, nonconstant η. Problem `square4()`, for which no exact solution is available, demonstrates a case where one side of the square is split into two intervals with different boundary conditions. Listing 26.1 shows the transcript of a sample session.

When an exact solution is available, the program computes and prints the errors in the L^∞ and L^2 norms. We don't compute the error in the energy norm since it requires an effort beyond the scope of this chapter. The formula (25.17) (page 347) is not applicable since the zero Dirichlet data was used in an essential way in its derivation.

26.5 ▪ The file *problem-spec.[ch]*

The file *problem-spec.h* remains unchanged from that in *Project Triangulate* of Chapter 23; therefore a symbolic link to that file will do. The file *problem-spec.c* defines the four boundary value problems depicted in Figures 26.2 and 26.3. The domain specifications are minor variations on the previous chapter's *square* domain. The novel element here

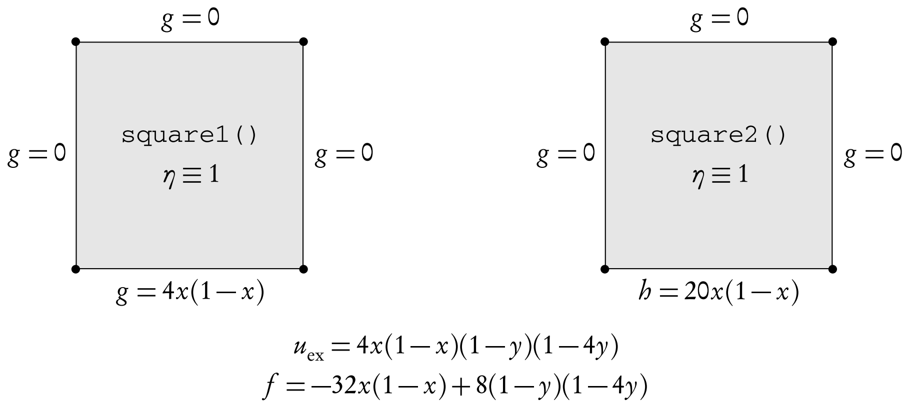


Figure 26.2: These figures declare two boundary value problems on the domain $\Omega = (0,1) \times (0,1)$. The version on the left, `square1()`, has Dirichlet data all around. The version on the right, `square2()`, has Neumann data on the bottom edge and Dirichlet data on the other three edges. In both cases the coefficient η is identically equal to 1 throughout, and the forcing function f and the exact solution u_{ex} are as given.

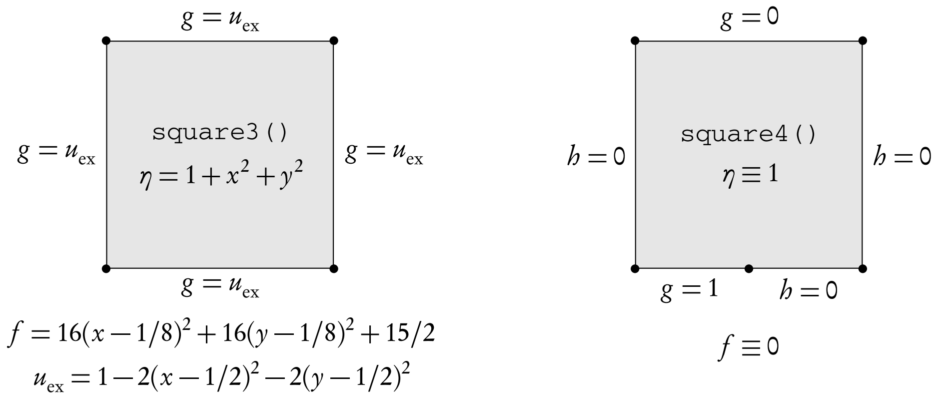


Figure 26.3: These figures declare two boundary value problems on the domain $\Omega = (0,1) \times (0,1)$. The version on the left, `square3()`, has Dirichlet data all around which is “inherited” from the exact solution u_{ex}. The coefficient η is nonconstant. The version on the right, `square4()`, has mixed Dirichlet and Neumann data on the various parts of the boundary. The coefficient η is identically equal to 1 throughout, and the forcing function f is zero.

is the care that goes into specifying the boundary condition types. Listing 26.2 shows the definition of the domain labeled `square2()` in Figure 26.2. We see that the bottom side is given the `FEM_BC_NEUMANN` attribute, while the other three sides are of the type `FEM_BC_DIRICHLET`. All four vertices, however, are of the type `FEM_BC_DIRICHLET` since the boundary values are prescribed on all four.

The functions `square_g()` and `square_h()` that appear in that listing may be defined in a variety of ways. Here is one possibility:

Listing 26.1: The transcript of a sample session with this chapter's program.

```
$ ./fem-demo
Usage: ./fem-demo d a
    d = the TWB quadrature strength
    a = maximal triangle area

$ ./fem-demo 10 0.002
domain is square1:
nodes = 428, edges = 1217, elems = 790
system stiffness matrix is 428x428 (=183184) has 2862 nonzero entries
errors: L^infty = 0.00977244, L^2 = 0.0022182
geomview output written to file square1.gv

domain is square2:
nodes = 428, edges = 1217, elems = 790
system stiffness matrix is 428x428 (=183184) has 2862 nonzero entries
errors: L^infty = 0.0111535, L^2 = 0.00219217
geomview output written to file square2.gv

domain is square3:
nodes = 428, edges = 1217, elems = 790
system stiffness matrix is 428x428 (=183184) has 2862 nonzero entries
errors: L^infty = 0.00378925, L^2 = 0.00196628
geomview output written to file square3.gv

domain is square4:
nodes = 433, edges = 1232, elems = 800
system stiffness matrix is 433x433 (=187489) has 2897 nonzero entries
geomview output written to file square4.gv
```

```
double square_g(double x, double y)
{
    return 0.0
}
```

We need not be concerned that `square_g()` returns incorrect values on `square2()`'s bottom edge (where the solution is nonzero) since g is never evaluated on that edge; it is evaluated only on the boundary parts of the `FEM_BC_DIRICHLET` type.

Another alternative, taking advantage of the availability of the exact solution, would be

```
double square_g(double x, double y)
{
    return square_u_exact(x, y);
}
```

since obviously the exact solution satisfies the boundary condition. The second alternative has a slight advantage over the first one since it applies to both the `square1()` and `square2()` problems of Figure 26.2.

For the same reasons, the function `square_h()` that produces the Neumann data for `square2()` may be given simply as

Listing 26.2: The specification in the file *problem-spec.c* of the domain corresponding to the boundary value problem `square2()` in Figure 26.2.

```
1  struct problem_spec *square2(void)
2  {
3      static struct problem_spec_point points[] =  {
4          { 0,    0.0,    0.0,    FEM_BC_DIRICHLET },
5          { 1,    1.0,    0.0,    FEM_BC_DIRICHLET },
6          { 2,    1.0,    1.0,    FEM_BC_DIRICHLET },
7          { 3,    0.0,    1.0,    FEM_BC_DIRICHLET },
8      };
9      static struct problem_spec_segment segments[] = {
10         { 0,    0,  1,  FEM_BC_NEUMANN },
11         { 1,    1,  2,  FEM_BC_DIRICHLET },
12         { 2,    2,  3,  FEM_BC_DIRICHLET },
13         { 3,    3,  0,  FEM_BC_DIRICHLET },
14     };
15     static struct problem_spec spec = {  // C99-style initialization!
16         .points      = points,
17         .npoints     = (sizeof points)/(sizeof points[0]),
18         .segments    = segments,
19         .nsegments   = (sizeof segments)/(sizeof segments[0]),
20         .holes       = NULL,
21         .nholes      = 0,
22         .f           = square_f,
23         .g           = square_g,
24         .h           = square_h,
25         .eta         = one,
26         .u_exact     = square_u_exact,
27     };
28     printf("domain is square2:\n");
29     return &spec;
30 }
```

```
double square_h(double x, double y)
{
    return 20*x*(1-x);
}
```

Again, we need not be concerned that it does not represent the correct Neumann data on edges other than the bottom; it is evaluated only on the bottom edge.

The domain specification of the problem `square4()` in Figure 26.3 is slightly different than the others since the bottom edge needs to be split into two distinct zones with `FEM_BC_DIRICHLET` and `FEM_BC_NEUMANN` attributes. To achieve that, define the square as a polygon with five vertices and five segments—the bottom edge consisting of two segments—and then specify the boundary types in the usual way.

26.6 ▪ The file *poisson.c*

This chapter's *poisson.c* is a slight modification of the previous chapter's. Listing 26.3 provides an outline. The function `error_and_exit()` remains unchanged. The functions `element_errors()` and `eval_errors()` are curtailed versions of the previous

Listing 26.3: An outline of the file *poisson.c*.

```
1  #include ...
2  ▶static void error_and_exit(int status,
3          const char *file, int line) ...
4  ▶static void apply_dirichlet_bc(struct elem *ep,
5          double (*g)(double x, double y), double k[3][4]) ...
6  ▶static void apply_neumann_bc(struct elem *ep,
7          double (*h)(double x, double y),
8          double k[3][4], struct Gauss_qdat *gqdat) ...
9  ▶static void compute_element_stiffness(
10         struct elem *ep, struct TWB_qdat *qdat,
11         double (*f)(double x, double y),
12         double (*eta)(double x, double y),
13         double k[3][4]) ...
14 ▶void poisson_solve(struct problem_spec *spec,
15         struct mesh *mesh, int d) ...
16 ▶static struct errors element_errors(struct problem_spec *spec,
17         struct TWB_qdat *qdat, struct elem *ep) ...
18 ▶struct errors eval_errors(struct problem_spec *spec,
19         struct mesh *mesh, int d)
```

code; they compute the error in the L^2 and L^∞ norms only. Computing the energy norm requires extra work, so we will not implement it. We will review the remaining functions in the following subsections.

26.6.1 ▪ The function `compute_element_stiffness()`

The function `compute_element_stiffness()` that appears on line 9 of Listing 26.3 requires only small adjustments relative to the previous chapter's function of the same name. First, we calculate the 12 entries of the 3×4 element stiffness matrix *exactly* as before. Next, in view of Remark 26.5 on page 366, we multiply the 3×3 part of the matrix (i.e., everything other than the rightmost column) by the average of η over the element. Since `compute_element_stiffness()` has already set up the TWB quadrature machinery for computing the fourth column of the element stiffness matrix, we may piggyback on that machinery to compute the integral of η. Very little extra code is needed here.

26.6.2 ▪ Applying the Neumann boundary data

According to the formulation (26.4), the contributions of the problem's force function, f, and the Neumann boundary data, h, appear as the sum $\int_\Omega f v_i \, dx + \int_{\Gamma_N} h v_i \, da$. At the element level, the first integral deposits entries in the fourth column of the 3×4 element stiffness matrix, just as it did in the previous chapter's program. The second integral deposits additional material in that column. The deposited values accumulate, reflecting the fact that the two integrals are being added.

To determine the contribution of the second integral at the element level, let us consider an element with vertices $\boldsymbol{v}_i$, $\boldsymbol{v}_j$, $\boldsymbol{v}_k$ (in counterclockwise order) whose edge i lies on the Γ_N part of the boundary. Recall that edge i lies opposite the vertex i; therefore it extends from the vertex $\boldsymbol{v}_j$ to the vertex $\boldsymbol{v}_k$. Figure 26.4 depicts a representative case.

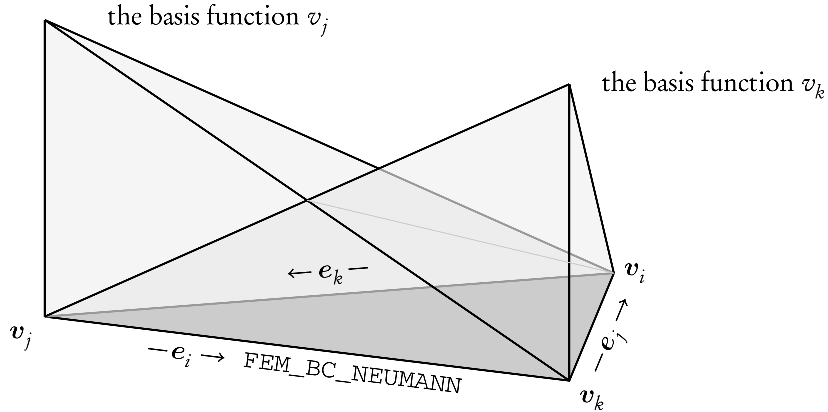


Figure 26.4: The diagram shows an element with vertices $\boldsymbol{v}_i$, $\boldsymbol{v}_j$, $\boldsymbol{v}_k$ and edge vectors $\boldsymbol{e}_i$, $\boldsymbol{e}_j$, $\boldsymbol{e}_k$. The edge $\boldsymbol{e}_i$ lies on the domain's boundary and has a boundary of type `FEM_BC_NEUMANN`. Parts of the basis functions v_j (centered at node $\boldsymbol{v}_j$) and v_k (centered at node $\boldsymbol{v}_k$) are shown. The trace of the basis function v_j on the edge $\boldsymbol{e}_i$ is a linear function that goes from 1 at the vertex $\boldsymbol{v}_j$ to 0 at the vertex $\boldsymbol{v}_k$. Similarly, the trace of the basis function v_k on the edge $\boldsymbol{e}_i$ is a linear function that goes from 1 at the vertex $\boldsymbol{v}_k$ to 0 at the vertex $\boldsymbol{v}_j$.

Additionally, the figure shows the fragments of the basis functions (pyramids) v_j and v_k centered at the vertices $\boldsymbol{v}_j$ and $\boldsymbol{v}_k$. We see in that figure that along the edge $\boldsymbol{e}_i$, the basis function v_j drops linearly from the value 1 at the endpoint v_j to the value 0 at the endpoint $\boldsymbol{v}_k$. To obtain an equation for that function, let us parametrize the edge $\boldsymbol{e}_i$ as in

$$\text{edge } \boldsymbol{e}_i: \quad \frac{1}{2}(1-\xi)\boldsymbol{v}_j + \frac{1}{2}(1+\xi)\boldsymbol{v}_k, \quad -1 \le \xi \le 1.$$

Thus, $\xi = -1$ puts us at $\boldsymbol{v}_j$, and $\xi = 1$ puts us at $\boldsymbol{v}_k$. Then the restriction of the basis function v_j to the edge $\boldsymbol{e}_i$ is the function $\frac{1}{2}(1-\xi)$ since it takes on the value 1 at the endpoint v_j and the value 0 at the endpoint $\boldsymbol{v}_k$. Consequently, the integral $\int_{\Gamma_N} h v_j \, da$ along the edge $\boldsymbol{e}_i$ takes the form

$$\frac{L}{2}\int_{-1}^{1} \frac{1}{2}(1-\xi)\, h\Big(\frac{1}{2}(1-\xi)\boldsymbol{v}_j + \frac{1}{2}(1+\xi)\boldsymbol{v}_k\Big)\, d\xi, \tag{26.10a}$$

where L is the actual length of the edge $\boldsymbol{e}_i$, that is, $L = ||\boldsymbol{v}_k - \boldsymbol{v}_j||$. The $L/2$ factor accounts for the change of the domain of integration from the edge $\boldsymbol{e}_i$ whose length is L to the parameter space $(-1,1)$ whose length is 2. (See equations (22.2) on page 291 for a clearer explanation.) The value of this integral is deposited in (that is, added to) the $(j,3)$ entry of the element stiffness matrix since it represents the $\int_{\Gamma_N} h v_j \, da$ integral.

This takes care of half of the required computation. We also need to evaluate the integral $\int_{\Gamma_N} h v_k \, da$ along the edge $\boldsymbol{e}_i$. You should have no difficulty in seeing that it is

Listing 26.4: An outline of the function apply_neumann_bc() in the file *poisson.c*.

```
1  static void apply_neumann_bc(struct elem *ep,
2          double (*h)(double x, double y),
3          double k[3][4], struct Gauss_qdat *gqdat)
4  {
5      for (int i = 0; i < 3; i++)               // scan the element's edges
6          if (ep→e[i]→bc == FEM_BC_NEUMANN) {
7              ▶ let (x1,y1) be the coords of one end of edge i
8              ▶ let (x2,y2) be the coords of the other end of edge i
9              ▶ compute the length L of the edge
10             double sum1 = 0.0;                // the integral in (26.10a)
11             double sum2 = 0.0;                // the integral in (26.10b)
12             while (gqdat→w ≠ -1) {            // do Gaussian quadrature
13                 ▶ ...
14                 gqdat++;
15             }
16             k[(i+1)%3][3] += L/2*sum1;        // deposit the compted values in
17             k[(i+2)%3][3] += L/2*sum2;        // the element stiffness matrix
18         }
19 }
```

given by

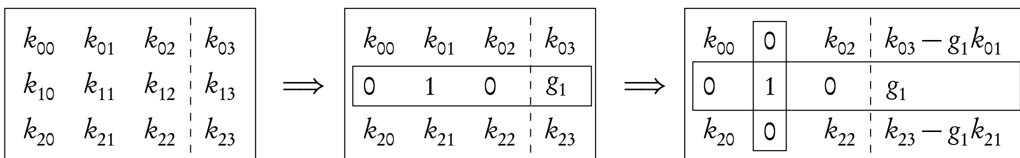
$$\frac{L}{2}\int_{-1}^{1}\frac{1}{2}(1+\xi)\,h\Big(\frac{1}{2}(1-\xi)\boldsymbol{v}_j+\frac{1}{2}(1+\xi)\boldsymbol{v}_k\Big)\,d\xi. \tag{26.10b}$$

The value of this integral is deposited in (that is, added to) the $(k,3)$ entry of the element stiffness matrix since it represents the $\int_{\Gamma_N} h v_k \, da$ integral.

The function apply_neumann_bc() that appears on line 6 of Listing 26.3 is responsible for evaluating these two integrals via Gaussian quadrature and depositing their values into the element stiffness matrix. Listing 26.4 gives an outline. Note that the indices of the vertices $\boldsymbol{v}_j$ and $\boldsymbol{v}_k$ appear as (i+1)%3 and (i+2)%3, as usual.

26.6.3 ▪ Applying the Dirichlet boundary data

The function apply_dirichlet_bc() that appears on line 4 in Listing 26.3 generalizes the previous chapter's enforce_zero_dirichlet_bc() function to nonzero Dirichlet data. It receives an element's 3×4 stiffness matrix k and modifies it to enforce the prescribed Dirichlet data in accordance with the algorithm described in formulas (26.6) and (26.7). Specifically, if the element's vertex i (where i is one of $0,1,2$) is of type FEM_BC_DIRICHLET, it zeros the k's row i, replaces k_{i3} with the prescribed boundary value, g_i, and sets k_{ii} to 1. The middle block of the diagram below illustrates this with $i=1$:



As was noted in connection with (26.7), we may use the modified row to eliminate the entries in column i through row operations and arrive at the matrix depicted at the right end of the diagram above. The elimination is not necessary and does not affect the problem's solution. However, it results in a sparser system stiffness matrix which is advantageous in connection with UMFPACK.

26.6.4 ▪ The order of the application of the boundary conditions

The rest of the file *poisson.c* is pretty much the same as Chapter 25's version. I wish to point out one subtle issue that you should take into account when making the transition to this chapter's version.

Let us look at Listing 25.4 of the function `poisson_solve()` on page 354. On line 30 we call `enforce_zero_dirichlet_bc()` to enforce the boundary conditions. In the current chapter that line will be replaced with a call to `apply_neumann_bc()` to process the problem's Neumann boundary data and then a call to `apply_dirichlet_bc()` to process the problem's Dirichlet boundary data. *The order of the calls is absolutely essential!* To see the reason why, consider a boundary node which is at the transition point between a Dirichlet edge and a Neumann edge. The application of the Neumann boundary condition *adds* the integrals (26.10) to that node's "force" term in the system stiffness matrix. The application of the Dirichlet boundary condition *resets* that force term to the prescribed boundary value. If the boundary conditions were applied, incorrectly, in the reversed order, then the value set by the Dirichlet data would be overwritten by the integrals of the Neumann data.

26.7 ▪ *Project FEM 2*

Complete the program, compile, and test with the four boundary value problems defined in Section 26.4. Compare your results to the transcript shown in that section and the graphs shown in Figure 26.5.

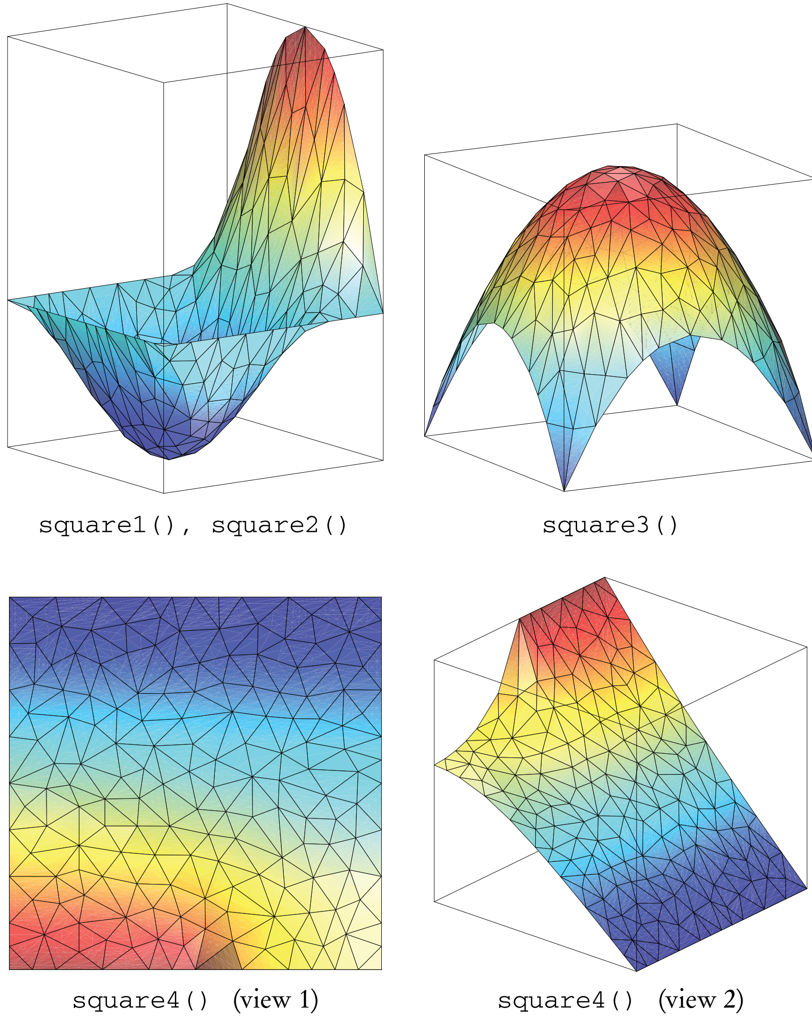


Figure 26.5: The graphs of the solutions of the four boundary value problems defined in Section 26.4 as obtained through the command `./fem-demo 10 0.005` and rendered in *Geomview*. The problems `square1()` and `square2()` have identical solutions. Two views of the solution of problem `square4()` are shown.

Appendix A
Barycentric coordinates

A.1 ▪ Barycentric coordinates

Figure A.1 shows the equilateral triangle $\mathscr{T}$ formed by the part of the plane $\lambda_1+\lambda_2+\lambda_3=1$ that lies in the positive octant in the $(\lambda_1,\lambda_2,\lambda_3)$ Cartesian coordinate system. It also shows a generic triangle with vertices $\boldsymbol{v}_1$, $\boldsymbol{v}_2$, $\boldsymbol{v}_3$ in the (x_1,x_2) Cartesian coordinate system. For reasons that will become clear shortly, we assume that the triangle T has a nonempty interior and that its vertices are enumerated in counterclockwise order. The equation

$$\boldsymbol{x}=\lambda_1\boldsymbol{v}_1+\lambda_2\boldsymbol{v}_2+\lambda_3\boldsymbol{v}_3 \tag{A.1}$$

establishes a bijection, i.e., a one-to-one and onto mapping, between the triangle $\mathscr{T}$ in the three-dimensional $(\lambda_1,\lambda_2,\lambda_3)$ space and the triangle T in the two-dimensional (x_1,x_2) space. In particular, the vertices $(1,0,0)$, $(0,1,0)$, and $(0,0,1)$ of $\mathscr{T}$ map to the vertices $\boldsymbol{v}_1$, $\boldsymbol{v}_2$, and $\boldsymbol{v}_3$ of T, respectively. The boundary of $\mathscr{T}$ maps onto the boundary of T, and the interior of $\mathscr{T}$ maps onto the interior of T. To every point $\boldsymbol{\lambda}=(\lambda_1,\lambda_2,\lambda_3)\in\mathscr{T}$ there corresponds a unique point $\boldsymbol{x}=(x_1,x_2)\in T$, and conversely, to every point $\boldsymbol{x}=(x_1,x_2)\in T$ there corresponds a unique point $\boldsymbol{\lambda}=(\lambda_1,\lambda_2,\lambda_3)\in\mathscr{T}$. The triplet $(\lambda_1,\lambda_2,\lambda_3)$ associated with the point of $\boldsymbol{x}\in T$ is called that point's *barycentric coordinates*. Figure A.1 illustrates the geometry.

To explore further, let us introduce the notation

$$\boldsymbol{v}_1=\begin{pmatrix}v_{11}\\ v_{12}\end{pmatrix},\quad \boldsymbol{v}_2=\begin{pmatrix}v_{21}\\ v_{22}\end{pmatrix},\quad \boldsymbol{v}_3=\begin{pmatrix}v_{31}\\ v_{32}\end{pmatrix},$$

whereby (A.1) takes the form

$$\begin{pmatrix}x_1\\ x_2\end{pmatrix}=\begin{pmatrix}v_{11} & v_{21} & v_{31}\\ v_{12} & v_{22} & v_{32}\end{pmatrix}\begin{pmatrix}\lambda_1\\ \lambda_2\\ \lambda_3\end{pmatrix},\qquad \text{where } \lambda_1+\lambda_2+\lambda_3=1,$$

or equivalently,

$$\begin{pmatrix}x_1\\ x_2\\ 1\end{pmatrix}=\begin{pmatrix}v_{11} & v_{21} & v_{31}\\ v_{12} & v_{22} & v_{32}\\ 1 & 1 & 1\end{pmatrix}\begin{pmatrix}\lambda_1\\ \lambda_2\\ \lambda_3\end{pmatrix}. \tag{A.2}$$

Equation (A.1), or its expanded form (A.2), carry us from the barycentric coordinates $(\lambda_1,\lambda_2,\lambda_3)$ to the Cartesian coordinates (x_1,x_2). The mapping in the reverse direction is

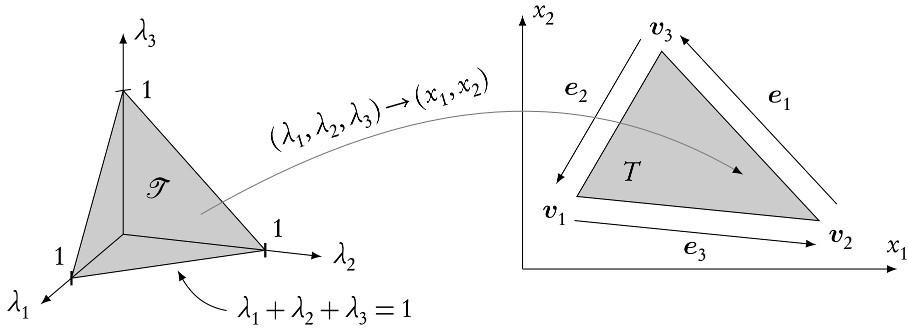


Figure A.1: The equilateral triangle $\mathscr{T}$ in the λ_1-λ_2-λ_3 coordinate system is mapped to the triangle T in the x-y coordinate system. The edge vectors $\boldsymbol{e}_1$, $\boldsymbol{e}_2$, $\boldsymbol{e}_3$ are defined in (A.3).

obtained by inverting the coefficient matrix. The outcome is best expressed in terms of the triangle T's *edge vectors*

$$\boldsymbol{e}_1 = \begin{pmatrix} e_{11} \\ e_{12} \end{pmatrix}, \quad \boldsymbol{e}_2 = \begin{pmatrix} e_{21} \\ e_{22} \end{pmatrix}, \quad \boldsymbol{e}_3 = \begin{pmatrix} e_{31} \\ e_{32} \end{pmatrix},$$

which are defined through

$$\boldsymbol{e}_1 = \boldsymbol{v}_3 - \boldsymbol{v}_2, \quad \boldsymbol{e}_2 = \boldsymbol{v}_1 - \boldsymbol{v}_3, \quad \boldsymbol{e}_3 = \boldsymbol{v}_2 - \boldsymbol{v}_1 \tag{A.3}$$

and shown graphically in Figure A.1. Upon inverting the mapping (A.2) we obtain

$$\begin{pmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \end{pmatrix} = \frac{1}{e_{11}e_{22} - e_{21}e_{12}} \begin{pmatrix} -e_{12} & e_{11} & (v_{21}v_{32} - v_{31}v_{22}) \\ -e_{22} & e_{21} & (v_{31}v_{12} - v_{11}v_{32}) \\ -e_{32} & e_{31} & (v_{11}v_{22} - v_{21}v_{12}) \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ 1 \end{pmatrix}. \tag{A.4}$$

Remark A.1. We recognize the expression in the denominator as the sole nonzero component of the cross product $\boldsymbol{e}_1 \times \boldsymbol{e}_2$—here we are thinking of the vectors $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$ as embedded in the three-dimensional space—therefore it equals the area of the parallelogram formed by those two vectors, that is to say, twice the area of the triangle T. Our assumptions that the triangle T has nonempty interior and that its vertices are ordered counterclockwise imply that $e_{11}e_{22} - e_{21}e_{12}$ is positive.

Since the triangle's area can be expressed equally well in terms of the cross products $\boldsymbol{e}_2 \times \boldsymbol{e}_3$ or $\boldsymbol{e}_3 \times \boldsymbol{e}_1$, any of the following expressions may be used to compute the area:

$$|T| = \frac{1}{2}(e_{11}e_{22} - e_{21}e_{12}) = \frac{1}{2}(e_{21}e_{32} - e_{31}e_{22}) = \frac{1}{2}(e_{31}e_{12} - e_{11}e_{32}). \tag{A.5}$$

Remark A.2. For future reference, let us note that (A.4) is equivalent to

$$\begin{pmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \end{pmatrix} = \frac{1}{2|T|} \begin{pmatrix} -e_{12} & e_{11} \\ -e_{22} & e_{21} \\ -e_{32} & e_{31} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \frac{1}{2|T|} \begin{pmatrix} v_{21}v_{32} - v_{31}v_{22} \\ v_{31}v_{12} - v_{11}v_{32} \\ v_{11}v_{22} - v_{21}v_{12} \end{pmatrix}. \tag{A.6}$$

Thus, by letting

$$B = \frac{1}{2|T|} \begin{pmatrix} -e_{12} & e_{11} \\ -e_{22} & e_{21} \\ -e_{32} & e_{31} \end{pmatrix}, \quad \boldsymbol{b} = \frac{1}{2|T|} \begin{pmatrix} v_{21}v_{32} - v_{31}v_{22} \\ v_{31}v_{12} - v_{11}v_{32} \\ v_{11}v_{22} - v_{21}v_{12} \end{pmatrix}, \tag{A.7}$$

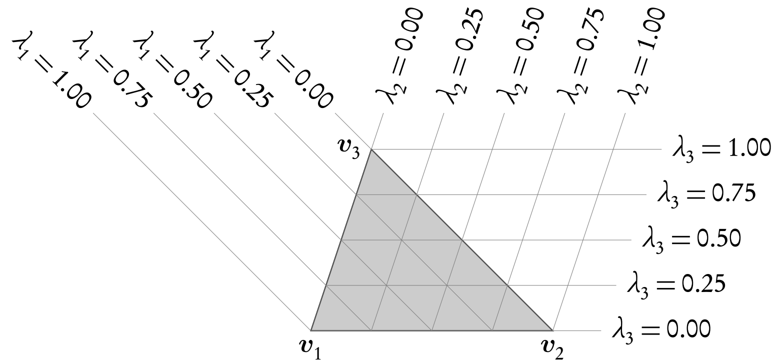


Figure A.2: The barycentric coordinate lines are parallel to the triangle's sides. The coordinate lines λ_i = constant are parallel to the edge opposite the vertex i.

and $\boldsymbol{\lambda} = (\lambda_1, \lambda_2, \lambda_3)$, (A.6) takes on the compact form

$$\boldsymbol{\lambda} = B\boldsymbol{x} + \boldsymbol{b}. \tag{A.8}$$

In conclusion, (A.1) maps from barycentric coordinates to Cartesian, and (A.8) maps from Cartesian to barycentric.

A.1.1 ▪ The barycentric coordinate lines

If you are new to the concept of barycentric coordinates, it would be worthwhile to spend a few minutes studying the diagram in Figure A.2, which shows the barycentric coordinate lines. Note, for instance, that the lines λ_i = constant are parallel to the edge opposite the vertex $\boldsymbol{v}_i$. In particular, the horizontal line marked $\lambda_3 = 0.25$ is parallel to the triangle's base. When we move along that line from the triangle's left edge to the right edge, the λ_2 coordinate varies from 0 to 0.75, and the λ_1 coordinate varies from 0.75 to 0. During that motion, and indeed anywhere in the triangle, we have $\lambda_1 + \lambda_2 + \lambda_3 = 1$.

A.1.2 ▪ The barycentric coordinates under a linear map

Consider a linear mapping L of the x_1-x_2 plane into itself. Suppose L takes the triangle T with vertices $\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3$ to a triangle T' with vertices $\boldsymbol{v}'_1, \boldsymbol{v}'_2, \boldsymbol{v}'_3$, as illustrated in Figure A.3. Consider a point $\boldsymbol{x} \in T$ specified through its barycentric coordinates $(\lambda_1, \lambda_2, \lambda_3)$, that is, $\boldsymbol{x} = \lambda_1\boldsymbol{v}_1 + \lambda_2\boldsymbol{v}_2 + \lambda_3\boldsymbol{v}_3$. Let $\boldsymbol{x}' = L\boldsymbol{x} \in T'$. Then

$$\begin{aligned} \boldsymbol{x}' = L\boldsymbol{x} &= L(\lambda_1\boldsymbol{v}_1 + \lambda_2\boldsymbol{v}_2 + \lambda_3\boldsymbol{v}_3) = \lambda_1 L\boldsymbol{v}_1 + \lambda_2 L\boldsymbol{v}_2 + \lambda_3 L\boldsymbol{v}_3 \\ &= \lambda_1\boldsymbol{v}'_1 + \lambda_2\boldsymbol{v}'_2 + \lambda_3\boldsymbol{v}'_3. \end{aligned}$$

We see that the barycentric coordinates of $\boldsymbol{x}'$ relative to T' are also $(\lambda_1, \lambda_2, \lambda_3)$. That is, *barycentric coordinates are preserved under linear maps*. This is a crucially important property in applications to integration on triangles.

A.2 ▪ Calculus on a triangle

Consider the triangles $\mathscr{T}$ and T shown in Figure A.1. As before, we assume that the vertices $\boldsymbol{v}_1$, $\boldsymbol{v}_2$, $\boldsymbol{v}_3$ are enumerated in counterclockwise order, and that T has positive

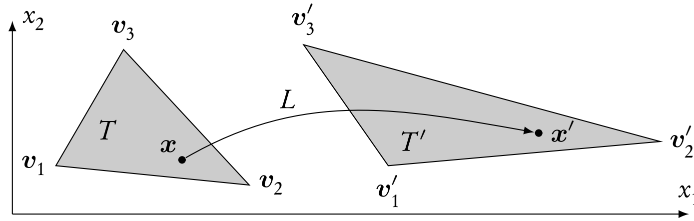


Figure A.3: The linear transformation L maps the triangle T to T'. The point $\boldsymbol{x}$ and its image $\boldsymbol{x}' = L\boldsymbol{x}$ have the same barycentric coordinates relative to T and T'.

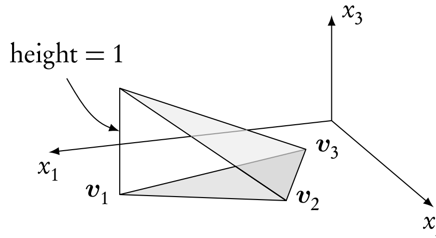


Figure A.4: The graph of the function $f(\boldsymbol{x}) = \tilde{f}(\boldsymbol{\lambda}(\boldsymbol{x}))$, where $\tilde{f}(\boldsymbol{\lambda}) = \lambda_1$.

area, that is, $|T| > 0$. We write $\boldsymbol{x}$ as a shorthand for (x_1, x_2) and $\boldsymbol{\lambda}$ as a shorthand for $(\lambda_1, \lambda_2, \lambda_3)$. In light of the one-to-one relationship between the Cartesian coordinates $\boldsymbol{x}$ and the barycentric coordinates $\boldsymbol{\lambda}$, we write $\boldsymbol{x}(\boldsymbol{\lambda})$ when we wish to emphasize the dependence of $\boldsymbol{x}$ on $\boldsymbol{\lambda}$. Similarly, we write $\boldsymbol{\lambda}(\boldsymbol{x})$ to emphasize the dependence of $\boldsymbol{\lambda}$ on $\boldsymbol{x}$. The explicit forms of these dependencies are given in (A.2) and (A.4).

A function $f : T \to \boldsymbol{R}$, through the change of variables $\boldsymbol{x} = \boldsymbol{x}(\boldsymbol{\lambda})$, induces a function $\tilde{f} : \mathscr{T} \to \boldsymbol{R}$ given by $\tilde{f}(\boldsymbol{\lambda}) = f(\boldsymbol{x}(\boldsymbol{\lambda}))$.

Example A.1. Consider the function $\tilde{f}(\boldsymbol{\lambda}) = \lambda_1$, where $\boldsymbol{\lambda} \in \mathscr{T}$. What does the corresponding function $f(\boldsymbol{x})$ look like on T? One way of figuring the answer is to observe that

$$\tilde{f}(1,0,0) = 1, \quad \tilde{f}(0,1,0) = 0, \quad \tilde{f}(0,0,1) = 0,$$

and therefore

$$f(\boldsymbol{v}_1) = 1, \quad f(\boldsymbol{v}_2) = 0, \quad f(\boldsymbol{v}_3) = 0.$$

Thus, the graph of f is a plane that passes through the vertices $\boldsymbol{v}_2$ and $\boldsymbol{v}_3$ and is sloped so that it reaches height 1 at vertex $\boldsymbol{v}_1$. Figure A.4 depicts the graph of $f(\boldsymbol{x})$.

Another way of figuring out the answer is to recall that the curves λ_1 = constant are straight lines parallel to the edge opposite the vertex $\boldsymbol{v}_1$. Therefore the graph of the function $f(\boldsymbol{x})$ has level curves parallel to the triangle T's $\boldsymbol{v}_2$–$\boldsymbol{v}_3$ edge which rise from level 0 near that edge to level 1 near the vertex $\boldsymbol{v}_1$ as λ_1 goes from 0 to 1. That, too, leads to Figure A.4.

Let us return to general functions of the form $f : T \to \boldsymbol{R}$, and $\tilde{f} : \mathscr{T} \to \boldsymbol{R}$, where $f(\boldsymbol{x}) = \tilde{f}(\boldsymbol{\lambda}(\boldsymbol{x}))$. According to (A.8), the derivative of the linear mapping $\boldsymbol{x} \mapsto \boldsymbol{\lambda}(\boldsymbol{x})$ is

the matrix B. Therefore the derivatives Df and $D\tilde{f}$ are related through the chain rule:

$$Df\Big|_{\boldsymbol{x}} = D\tilde{f}\Big|_{\boldsymbol{\lambda}(\boldsymbol{x})} B. \tag{A.9}$$

Remark A.3. For the purposes of the chain rule, the derivatives Df and $D\tilde{f}$ are viewed as *single-row matrices*:

$$Df = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}\right), \qquad D\tilde{f} = \left(\frac{\partial \tilde{f}}{\partial \lambda_1}, \frac{\partial \tilde{f}}{\partial \lambda_2}, \frac{\partial \tilde{f}}{\partial \lambda_3}\right).$$

As a consistency check, let us note that in (A.9) Df is 1×2, $D\tilde{f}$ is 1×3, and B is 3×2. In this connection let us also note that the gradient of a function is usually deemed a *column vector*, that is, $\nabla f = (Df)^T$.

Example A.2. Consider the function $\tilde{f}(\boldsymbol{\lambda}) = \lambda_1$, and let $f(\boldsymbol{x}) = \tilde{f}\big(\boldsymbol{\lambda}(\boldsymbol{x})\big)$. The graph of f is shown in Figure A.4. We have $D\tilde{f} = (1,0,0)$, and therefore from (A.9) we get

$$Df\Big|_{\boldsymbol{x}} = \begin{pmatrix}1 & 0 & 0\end{pmatrix}\left[\frac{1}{2|T|}\begin{pmatrix}-e_{12} & e_{11}\\ -e_{22} & e_{21}\\ -e_{32} & e_{31}\end{pmatrix}\right] = \frac{1}{2|T|}\begin{pmatrix}-e_{12} & e_{11}\end{pmatrix},$$

whence

$$\nabla f = \frac{1}{2|T|}\begin{pmatrix}-e_{12}\\ e_{11}\end{pmatrix}.$$

Note that the gradient vector is perpendicular to the edge vector $\boldsymbol{e}_1 = \binom{e_{11}}{e_{12}}$, as it should be, since f's level curves are parallel to $\boldsymbol{e}_1$.

Example A.3. Generalizing the previous example, let $\tilde{\phi}_i(\boldsymbol{\lambda}) = \lambda_i$ for $i = 1,2,3$, and let $\phi_i(\boldsymbol{x}) = \tilde{\phi}_i(\boldsymbol{\lambda}(\boldsymbol{x}))$. Then

$$\nabla \phi_i = \frac{1}{2|T|}\begin{pmatrix}-e_{i2}\\ e_{i1}\end{pmatrix},$$

and therefore

$$\nabla\phi_i \cdot \nabla\phi_j = \frac{1}{4|T|^2}\begin{pmatrix}-e_{i2}\\ e_{i1}\end{pmatrix}\cdot\begin{pmatrix}-e_{j2}\\ e_{j1}\end{pmatrix} = \frac{1}{4|T|^2}(e_{i1}e_{j1} + e_{i2}e_{j2}) = \frac{1}{4|T|^2}\boldsymbol{e}_i\cdot\boldsymbol{e}_j. \tag{A.10}$$

This result plays a crucial role in the calculation of element stiffness matrices in Chapters 25 and 26.

Bibliography

[1] Robert A. Adams and John J. F. Fournier. *Sobolev spaces*, volume 140 of *Pure and Applied Mathematics*. Elsevier/Academic Press, Amsterdam, second edition, 2003. (Cited on p. 359)

[2] Nicholas D. Alikakos and Rouben Rostamian. Lower bound estimates and separable solutions for homogeneous equations of evolution in Banach space. *Journal of Differential Equations*, 43(3):323–344, 1982. (Cited on p. 285)

[3] Grégoire Allaire. *Shape optimization by the homogenization method*, volume 146 of *Applied Mathematical Sciences*. Springer-Verlag, New York, 2002. (Cited on p. 215)

[4] Kendall E. Atkinson. *An Introduction to Numerical Analysis*. Wiley, New York, second edition, 1989. (Cited on pp. 137, 292, 295, 296)

[5] O. Axelsson and V. A. Barker. *Finite element solution of boundary value problems*, volume 35 of *Classics in Applied Mathematics*. SIAM, Philadelphia, 2001. Theory and Computation, Reprint of the 1984 original. (Cited on pp. 342, 360)

[6] R. Barrett, M. W. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, Philadelphia, 1993. (Cited on p. 79)

[7] Conrad Barski. *Land of Lisp: Learn to Program in Lisp, One Game at a Time!* No Starch Press, San Francisco, CA, 2011. (Cited on p. 161)

[8] Martin P. Bendsøe and Carlos A. Mota Soares, editors. *Topology design of structures*, volume 227 of *NATO Advanced Science Institutes Series E: Applied Sciences*. Kluwer Academic Publishers Group, Dordrecht, The Netherlands, 1993. (Cited on p. 215)

[9] M. Bertsch and R. Rostamian. The principle of linearized stability for a class of degenerate diffusion equations. *Journal of Differential Equations*, 57(3):373–405, 1985. (Cited on p. 285)

[10] Dietrich Braess. *Finite Elements*. Cambridge University Press, Cambridge, UK, third edition, 2007. Theory, fast solvers, and applications in elasticity theory, Translated from the German by Larry L. Schumaker. (Cited on pp. 342, 345, 359)

[11] Susanne C. Brenner and L. Ridgway Scott. *The mathematical theory of finite element methods*, volume 15 of *Texts in Applied Mathematics*. Springer, New York, third edition, 2008. (Cited on pp. 342, 360)

[12] Philippe G. Ciarlet. *The finite element method for elliptic problems*, volume 40 of *Classics in Applied Mathematics*. SIAM, Philadelphia, 2002. Reprint of the 1978 original. (Cited on pp. 342, 360, 364)

[13] Ronald Cools. An encyclopaedia of cubature formulas. *Journal of Complexity*, 19(3):445–453, 2003. (Cited on p. 321)

[14] T. A. Davis. A column pre-ordering strategy for the unsymmetric-pattern multifrontal method. *ACM Transactions on Mathematical Software*, 30(2):165–195, 2004. (Cited on p. 85)

[15] T. A. Davis and I. S. Duff. A combined unifrontal/multifrontal method for unsymmetric sparse matrices. *ACM Transactions on Mathematical Software*, 25(1):1–19, 1999. (Cited on p. 85)

[16] Timothy A. Davis. *Direct methods for sparse linear systems*, volume 2 of *Fundamentals of Algorithms*. SIAM, Philadelphia, 2006. (Cited on pp. 80, 85)

[17] A. K. Dewdney. Simulated Evolution: Wherein bugs learn to hunt bacteria. *Scientific American*, 260(5):138–141, 1989. (Cited on p. 161)

[18] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct methods for sparse matrices*. Monographs on Numerical Analysis. The Clarendon Press, Oxford University Press, New York, 1986. (Cited on pp. 79, 80)

[19] Lawrence C. Evans. *Partial differential equations*, volume 19 of *Graduate Studies in Mathematics*. American Mathematical Society, Providence, RI, second edition, 2010. (Cited on p. 359)

[20] Stanley J. Farlow. *Partial Differential Equations for Scientists and Engineers*. Dover Publications Inc., New York, 1993. Revised reprint of the 1982 original. (Cited on p. 338)

[21] Avner Friedman. *Partial Differential Equations of Parabolic Type*. Prentice–Hall, Englewood Cliffs, NJ, 1964. (Cited on p. 252)

[22] David Gilbarg and Neil S. Trudinger. *Elliptic partial differential equations of second order*. Classics in Mathematics. Springer-Verlag, Berlin, 2001. Reprint of the 1998 edition. (Cited on p. 359)

[23] Mark S. Gockenbach. *Understanding and Implementing the Finite Element Method*. SIAM, Philadelphia, 2006. (Cited on pp. 342, 345, 359)

[24] Alfred Haar. Zur theorie der orthogonalen funktionensysteme. *Mathematische Annalen*, 69:331–371, 1910. (Cited on pp. 93, 94, 382)

[25] Alfred Haar. On the theory of orthogonal function systems. In Christopher Heil and David F. Walnut, editors, *Fundamental Papers in Wavelet Theory*, pages 155–188. Princeton University Press, Princeton, NJ, 2006. (English translation of [24]; Georg Zimmermann, translator). (Cited on p. 93)

[26] David R. Hanson. *C Interfaces and Implementations: Techniques for Creating Reusable Software*. Addison–Wesley, Boston, MA, 1997. (Cited on pp. 11, 152)

[27] Richard Heathfield, Lawrence Kirby, et al. *C Unleashed*. Sams Publishing, Indianapolis, IN, 2000. (Cited on p. 48)

[28] F. B. Hildebrand. *Introduction to Numerical Analysis*. McGraw–Hill, New York, second edition, 1974. (Cited on pp. 292, 295, 296)

[29] Eugene Isaacson and Herbert Bishop Keller. *Analysis of Numerical Methods*. Dover Publications Inc., New York, 1994. Corrected reprint of the 1966 original. (Cited on p. 253)

[30] C. T. Kelley. Detection and remediation of stagnation in the Nelder–Mead algorithm using a sufficient decrease condition. *SIAM Journal on Optimization*, 10(1):43–55, 1999. (Cited on p. 197)

[31] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice–Hall, Englewood Cliffs, NJ, first edition, 1978. (Cited on p. 4)

[32] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice–Hall, Englewood Cliffs, NJ, second edition, 1988. (Cited on pp. 4, 7)

[33] David Kincaid and Ward Cheney. *Numerical Analysis: Mathematics of Scientific Computing*. American Mathematical Society, Providence, RI, third edition, 2002. (Cited on pp. 137, 253, 255, 258)

[34] Donald E. Knuth. *The Art of Computer Programming: Sorting and Searching*. Addison–Wesley, Reading, MA, second edition, 1998. (Cited on p. 155)

[35] Stephen G. Kochan. *Programming in C*. Sams Publishing, Indianapolis, IN, third edition, 2005. (Cited on pp. xv, 7)

[36] Tamara G. Kolda, Robert Michael Lewis, and Virginia Torczon. Optimization by direct search: New perspectives on some classical and modern methods. *SIAM Review*, 45(3):385–482, 2003. (Cited on p. 197)

[37] V. A. Kozlov, V. G. Maz'ya, and J. Rossmann. *Elliptic boundary value problems in domains with point singularities*, volume 52 of *Mathematical Surveys and Monographs*. American Mathematical Society, Providence, RI, 1997. (Cited on p. 359)

[38] Erwin Kreyszig. *Advanced Engineering Mathematics*. John Wiley & Sons Inc., New York, tenth edition, 2011. (Cited on p. 338)

[39] O. A. Ladyzhenskaya. *The boundary value problems of mathematical physics*, volume 49 of *Applied Mathematical Sciences*. Springer-Verlag, New York, 1985. Translated from the Russian by Jack Lohwater. (Cited on p. 359)

[40] Jeffrey C. Lagarias, James A. Reeds, Margaret H. Wright, and Paul E. Wright. Convergence properties of the Nelder–Mead simplex method in low dimensions. *SIAM Journal on Optimization*, 9(1):112–147, 1998. (Cited on pp. 194, 197)

[41] J.-L. Lions and E. Magenes. *Non-homogeneous boundary value problems and applications. Volumes* I–III. Springer-Verlag, New York, 1972. Translated from the French by P. Kenneth, Die Grundlehren der mathematischen Wissenschaften, Band 181. (Cited on p. 359)

[42] J. N. Lyness and Ronald Cools. A survey of numerical cubature over triangles. In *Mathematics of Computation* 1943–1993*: A half-century of computational mathematics (Vancouver, BC,* 1993*)*, volume 48 of *Proceedings of Symposia in Applied Mathematics*, pages 127–150. American Mathematical Society, Providence, RI, 1994. (Cited on p. 321)

[43] R. M. M. Mattheij, S. W. Rienstra, and J. H. M. ten Thije Boonkkamp. *Partial differential equations*. SIAM Monographs on Mathematical Modeling and Computation. SIAM, Philadelphia, 2005. Modeling, analysis, computation. (Cited on pp. 338, 359)

[44] Vladimir Maz'ya and Jürgen Rossmann. *Elliptic equations in polyhedral domains*, volume 162 of *Mathematical Surveys and Monographs*. American Mathematical Society, Providence, RI, 2010. (Cited on p. 359)

[45] K. I. M. McKinnon. Convergence of the Nelder–Mead simplex method to a nonstationary point. *SIAM Journal on Optimization*, 9(1):148–158, 1998. (Cited on p. 197)

[46] Carl D. Meyer. *Matrix Analysis and Applied Linear Algebra*. SIAM, Philadelphia, 2000. (Cited on p. 213)

[47] Lee Middleton and Jayanthi Sivaswamy. *Hexagonal Image Processing: A Practical Approach*. Springer-Verlag, London, 2005. (Cited on p. 113)

[48] Arch W. Naylor and George R. Sell. *Linear operator theory in engineering and science*, volume 40 of *Applied Mathematical Sciences*. Springer-Verlag, New York, 1982. (Cited on p. 94)

[49] J. A. Nelder and R. Mead. A simplex method for function minimization. *The Computer Journal*, 7(4):308–313, 1965. (Cited on pp. 193, 196, 211, 212)

[50] J. A. Nelder and R. Mead. A simplex method for function minimization: Errata. *The Computer Journal*, 8(1):27, 1965. (Cited on p. 193)

[51] Yves Nievergelt. *Wavelets Made Easy*. Birkhäuser, Boston, MA, 2001. 2nd printing with corrections. (Cited on pp. 93, 102)

[52] Ján Plesník. Finding the orthogonal projection of a point onto an affine subspace. *Linear Algebra and Its Applications*, 422(2-3):455–470, 2007. (Cited on p. 213)

[53] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C*. Cambridge University Press, Cambridge, UK, second edition, 1992. (Cited on pp. 73, 93, 193, 196, 197, 296)

[54] Alfio Quarteroni. *Numerical models for differential problems*, volume 2 of *MS&A. Modeling, Simulation and Applications*. Springer-Verlag Italia, Milan, 2009. Translated from the 4th (2008) Italian edition by Silvia Quarteroni. (Cited on p. 359)

[55] Michael Renardy and Robert C. Rogers. *An introduction to partial differential equations*, volume 13 of *Text in Applied Mathematics*. Springer, New York, 1993. (Cited on p. 359)

[56] Walter Rudin. *Functional analysis*. International Series in Pure and Applied Mathematics. McGraw–Hill, New York, second edition, 1991. (Cited on p. 359)

[57] Ridgeway Scott. Optimal L^∞ estimates for the finite element method on irregular meshes. *Mathematics of Computation*, 30(136):681–697, 1976. (Cited on p. 345)

[58] Thomas I. Seidman. On the stability of certain difference schemes. *Numerische Mathematik*, 5:201–210, 1963. (Cited on pp. 260, 262)

[59] Yair Shapira. *Solving PDEs in C++*, volume 9 of *Computational Science & Engineering*. SIAM, Philadelphia, second edition, 2012. Numerical Methods in a Unified Object-Oriented Approach. (Cited on p. 360)

[60] Jonathan Richard Shewchuk. Triangle: Engineering a 2D quality mesh generator and Delaunay triangulator. In Ming C. Lin and Dinesh Manocha, editors, *Applied Computational Geometry: Towards Geometric Engineering*, volume 1148 of *Lecture Notes in Computer Science*, pages 203–222. Springer-Verlag, Berlin, May 1996. (From the First ACM Workshop on Applied Computational Geometry). (Cited on p. 301)

[61] Jonathan Richard Shewchuk. Delaunay refinement algorithms for triangular mesh generation. *Computational Geometry: Theory and Applications*, 22(1–3):21–74, 2002. (Cited on p. 301)

[62] Ana Maia Soane, Manil Suri, and Rouben Rostamian. The optimal convergence rate of a C^1 finite element method for non-smooth domains. *Journal of Computational and Applied Mathematics*, 233(10):2711–2723, 2010. (Cited on p. 359)

[63] Ana Maria Soane and Rouben Rostamian. Variational problems in weighted Sobolev spaces on non-smooth domains. *Quarterly of Applied Mathematics*, 68(3):439–458, 2010. (Cited on p. 359)

[64] Olaf Steinbach. *Numerical Approximation Methods for Elliptic Boundary Value Problems*. Springer, New York, 2008. Finite and Boundary Elements. (Cited on p. 359)

[65] James Stewart. *Multivariable Calculus*. Brooks/Cole, Belmont, CA, 2008. (Cited on pp. 290, 339)

[66] Eric J. Stollnitz, Tony D. DeRose, and David H. Salesin. Wavelets for computer graphics: A primer, part 1. *IEEE Computer Graphics and Applications*, 15(3):76–84, 1995. (Cited on p. 93)

[67] Eric J. Stollnitz, Tony D. DeRose, and David H. Salesin. Wavelets for computer graphics: A primer, part 2. *IEEE Computer Graphics and Applications*, 15(4):75–85, 1995. (Cited on p. 93)

[68] Eric J. Stollnitz, Tony D. DeRose, and David H. Salesin. *Wavelets for Computer Graphics: Theory and Applications*. Morgan Kaufmann Publishers, San Francisco, CA, 1996. (Cited on pp. 93, 102)

[69] Gilbert Strang. Wavelets and dilation equations: A brief introduction. *SIAM Review*, 31:614–627, 1989. (Cited on p. 93)

[70] Gilbert Strang. Wavelets. *American Scientist*, 82:250–255, 1994. (Cited on p. 93)

[71] John C. Strikwerda. *Finite Difference Schemes and Partial Differential Equations*. SIAM, Philadelphia, second edition, 2004. (Cited on pp. 253, 262)

[72] Barna Szabó and Ivo Babuška. *Finite Element Analysis*. John Wiley & Sons Inc., New York, 1991. (Cited on p. 359)

[73] Gábor Szegő. *Orthogonal Polynomials*. American Mathematical Society, Providence, RI, fourth edition, 1975. American Mathematical Society, Colloquium Publications, Vol. XXIII. (Cited on p. 295)

[74] Mark A. Taylor, Beth A. Wingate, and Len P. Bos. A cardinal function algorithm for computing multivariate quadrature points. *SIAM Journal on Numerical Analysis*, 45(1):193–205, 2007. See [75] for the full-length article. (Cited on pp. 319, 321, 385)

[75] Mark. A. Taylor, Beth A. Wingate, and Len P. Bos. Several new quadrature formulas for polynomial integration in the triangle. *arXiv*,
`<http://arxiv.org/abs/math/0501496v2>`,
February 2007, pages 1–14. Quadrature data tables, bundled with the article's LaTeX source, are available electronically from this site. An abbreviated version of this article appeared in [74]. (Cited on pp. 319, 321, 322, 323, 385)

[76] Virginia Torczon. On the convergence of pattern search algorithms. *SIAM Journal on Optimization*, 7(1):1–25, 1997. (Cited on p. 197)

[77] Juan Luis Vázquez. *The porous medium equation*. Oxford Mathematical Monographs. The Clarendon Press, Oxford University Press, Oxford, UK, 2007. Mathematical theory. (Cited on pp. 283, 285)

[78] S. Wandzura and H. Xiao. Symmetric quadrature rules on a triangle. *Computers & Mathematics with Applications*, 45(12):1829–1840, 2003. (Cited on p. 321)

[79] H. F. Weinberger. *A first course in partial differential equations with complex variables and transform methods*. Dover Publications Inc., New York, 1995. Corrected reprint of the 1965 original. (Cited on p. 338)

[80] Margaret H. Wright. Direct search methods: Once scorned, now respectable. In F. Griffiths, D and G. A. Watson, editors, *Proceedings of the* 1995 *Dundee Biennial Conference in Numerical Analysis*, Numerical Analysis 1995, pages 191–208. Longman, Harlow, UK. (Cited on p. 196)

[81] C. Ray Wylie and Louis C. Barrett. *Advanced Engineering Mathematics*. McGraw–Hill, New York, sixth edition, 1995. (Cited on p. 338)

[82] Hong Xiao and Zydrunas Gimbutas. A numerical algorithm for the construction of efficient quadrature rules in two and higher dimensions. *Computers & Mathematics with Applications*, 59(2):663–676, 2010. (Cited on p. 321)

[83] E. C. Zachmanoglou and Dale W. Thoe. *Introduction to Partial Differential Equations with Applications*. Dover Publications Inc., New York, second edition, 1986. Reprint of the 1976 original. (Cited on p. 338)

[84] A. Zygmund. *Trigonometric Series*, volume I. Cambridge University Press, New York, second edition, 1959. (Cited on p. 93)

Index

$ (the Unix shell prompt), 13
& (the "address-of" operator), 19
* (dereferencing operator), 20
< (redirect stream), 16
>> (append to file), 16
> (redirect stream), 16
@ (Unix symlink decoration), 14
\0 (ASCII NUL character), 24, 66
\ (Unix command-line continuation), 31
\ (line continuation in make), 34
| (pipe stream), 16
$\neq$ (same as !=), 11
$\rightarrow$ (same as ->), 11
$\leq$ (same as <=), 11
$\geq$ (same as >=), 11
#include guards, 11, 51
    leading underscores in, 51

abs(), 140
absolute error, 137
add_plants(), 184
affine constraints, 211, 247
affine space, 211
affine-preserving, 211
animal, 164
animal death, 164
animal direction, 164
animal energy spent, 164
animal reproduction, 164
animal turning, 164
animate (image animator), 190
anisotropic material, 362
annulus(), 308, 315, 332
annulus_f(), 330, 350
apply_dirichlet_bc(), 370, 372
apply_neumann_bc(), 370, 372
area of triangle, 376
argc, 25
argv, 25
array, 21
    length of, 22
    name decays to pointer, 22, 25
ASCII NUL character, 24, 66
assembling (in FEM), 343, 344
atof(), 29
atoi(), 29
atol(), 29
atoll(), 29

back substitution, 275
Barenblatt, *see* porous medium equation
barycentric coordinates, 343, 375
bisection algorithm, 137, 140
bitmap images, 73
boundary value problem, 302, 337, 339, 361
buffer, 63

C
    C11, 5
    C89, 4
    C99, 4
    compilers, 7
    extent of, in this book, 5
    infringements into C99, 5
cantilever truss, 222
chromosome, 161, 164
clip_matrix(), 138, 141
clone(), 187
color image, 117
command-line arguments, 25
comparison function, 153
complete orthonormal basis, 94
Compressed Column Storage (CCS), 79–81, 85, 353
compute_element_stiffness(), 352, 370
conductivity, 362
cons cell, 148
conscell data type, 148
convert (image conversion program), 190
Crank–Nicolson, 258

cubic equation
- explicit solution, 286
- implementation, 288

Cygwin, 7

dead_or_alive(), 182
deformed configuration, 215
degenerate parabolic equation, 283
dereferencing, 20
diffusion process, 362
diffusivity, 362
dimension (of a linear space), 94, 95
Dirichlet boundary condition, 302
Dirichlet boundary data, 361
display (image viewer), 190
divergence, 338
divergence theorem, 338
do_demo(), 315, 331, 332
done(), 204
driver, 9

Eden, 162, 163
- center cell formula, 164
- height and width, 166
- location, 163, 166
- location formula, 164

edge vector(s), 312, 376
elasticity, one-dimensional, 217
element_errors(), 356, 358, 369, 370
enforce_zero_dirichlet_bc(), 351
error function (erf()), 264, 272
error_and_exit(), 351, 369, 370
eval_errors(), 351, 357, 359, 369, 370
eval_f(), 333, 334
evaluate_reactions(), 243, 247
evaluate_stresses(), 242, 247
evince (PS, EPS, and PDF viewer), 190
evolution, 161
- command-line, 166
- long run behavior, 165
- snapshots, 162, 167

evolve(), 188
evolve_with_figs(), 188
EXIT_FAILURE, 46
EXIT_SUCCESS, 48
explicit scheme, 253, 254, 260

fabs(), 140
feed(), 186
feh (image viewer), 190
FEM_BC_DIRICHLET, 305, 352, 367–369, 372
FEM_BC_NEUMANN, 305, 367, 369, 371, 372
fetch-line module, 63
fetch_line(), 169, 170
fetch_line_aux(), 170
fgetc(), 15
fgets(), 15, 63
__FILE__, 46, 50
file organization, 13
fill ratio, 74
filter (a linked list), 157
finite difference, 251, 283
- backward difference, 252
- conditionally stable, 256
- Crank–Nicolson, 258
- explicit scheme, 253, 254, 260
- forward difference, 252
- get_error(), 277
- heat1(), 270
- heat_implicit(), 272, 277
- implicit scheme, 256
- plot_curve(), 275
- Seidman sweep, 260
- stable, 256
- stencil, 254
- test problems, 263
- trisolve(), 274, 280
- unconditionally stable, 258–260, 262
- unstable, 256

Finite Element Method (FEM), 337, 361
- boundary conditions, 344
- element, 343
- element force vector, 343
- element stiffness matrix, 343, 365–366
- error analysis, 345
- error in energy norm, 347
- error norms, 345
- global vertex numbers, 343
- h (mesh size), 345
- local vertex numbers, 343
- nodal shape functions, 343
- system stiffness matrix, 341, 365

finite speed of propagation, 284
flux vector, 361
food, 164
- in the Eden, 164
- in the world, 164

for-loop initialization, 5
Fourier series, 93
- generalized, 94
- pointwise convergence, 93
- uniform convergence, 93

fprintf(), 15
fputc(), 15
fputs(), 15
free(), 47
free_annulus(), 311, 315, 332
free_dmatrix(), 57
free_herd(), 182

free_image(), 129
free_links_list(), 237
free_matrix(), 58
free_mesh(), 313, 315
free_nodes_list(), 237
free_truss(), 231, 239, 249
free_vector(), 55
fscanf(), 15
__func__, 279

Galerkin approximation, 340, 364
Garden of Eden, 162
gauss_qdat(), 298
Gaussian elimination, 86
Gaussian quadrature, 291, 372
  table sentinel, 298, 299
  weights, 296
gcc, 8, 31
gcc flags
  -MM, 33, 36
  -O2, 8
  -Wall, 8
  -std=c89, 8
  -std=c99, 8
gene, 161, 164
  activation, 161, 164
  dominant, 161, 162, 164
  probability of selection, 164
gene_to_activate(), 184
generalized Fourier series, 94
Geomview, 7, 8, 266, 275, 287, 332, 333, 350, 366
get_animal_specs(), 170
get_centroid(), 202
get_eden_dimens(), 170
get_error(), 277
get_links(), 232, 237
get_loads(), 232, 238
get_nodes(), 231, 234
get_plant_energy(), 170
get_reproduction_threshold_energy(), 170
get_world_dimens(), 169
getchar(), 15
gradient, 338
Gram–Schmidt orthogonalization, 294
grayscale image, 116
Green's identity, 339
Green's Theorem, 339
gv (PS and EPS viewer), 190

Haar scaling functions, 96
Haar wavelet basis, 100
Haar wavelet coefficients, 100
Haar wavelet transform, 102
Haar wavelets, 93, 96
haar_transform_matrix(), 105
haar_transform_matrix_forward(), 107
haar_transform_matrix_reverse(), 109
haar_transform_vector(), 105
haar_transform_vector_forward(), 105
haar_transform_vector_reverse(), 107
header guards, *see* #include guards
heat conduction, 362
heat equation, 251
heat_implicit(), 272, 277
hexadecimal notation, 19
Hilbert matrix, 57, 58
homogeneous material, 362

image
  viewing EPS, 9
  viewing PGM and PPM, 9
image analysis with wavelets, 135
image header, 120
image I/O, 113
image-io module, 121, 124
image-io-test-0.c, 129
ImageMagick, 190
implementation, 10
implicit scheme, 256
include guards, *see* #include guards
infinitesimal strain, 220
inhomogeneous material, 362
initialize_plants(), 183
inline declaration specifier, 72
inline function specifier, 7
inner product, 93, 294
integrate_over_triangle(), 333
integration on triangles, 319
interface, 10
inverse Haar wavelet transform, 102
isomorphism, 98
isotropic material, 362
isspace(), 66

kitchen analogy, 10

$L^2(0,1)$, 93
Lagrange interpolation, 292
Laplacian, 338
leak memory, 47
Legendre polynomials, 294
  recursion formulas, 295
less (Unix pager), 17
libnetpbm library, 9, 118
__LINE__, 46, 50
link_stretch(), 242
linked list, 147
  traversing, 147
linking, 32

ll_append(), 160
ll_filter(), 157, 159, 182
ll_free(), 151, 182, 237
ll_length(), 159
ll_map(), 160
ll_pop(), 150
ll_push(), 148, 174, 187
ll_reverse(), 152, 233
ll_sort(), 153, 181, 183
ln (make a symlink in Unix), 13
lrint(), 141
ls (list files in Unix), 13
ls -F (list files in Unix), 13
LU factorization, 86

magic number, 73, 115, 116
main() (declaration), 25
make, 31
  # (comment character), 37
  $(CC), 37
  $(CFLAGS), 35
  $@, 37
  $^, 37
  \ (line continuation), 34
  clean, 39
  tab preceding a command, 34
  link with libraries, 40
make_3array(), 61
make_4array(), 61
make_dmatrix(), 56
make_dvector(), 54
make_ivector(), 54
make_link(), 238
make_matrix(), 58
make_matrix_loop_counter, 58
make_mesh(), 313, 332
make_node(), 236
make_vector(), 54
mkdir (make directory in Unix), 13
malloc(), 45
  allocating zero bytes, 52
malloc_or_exit(), 49
man pages in Unix, 140
matrix, row-wise vs. column-wise sweep, 202
maximal triangle area, 303, 313
memory leak, 47
menu analogy, 10
mesh_to_eps(), 315
meshing, 301, 342
metric, 94
module, 9
move(), 185
multidimensional arrays, 23
mutate(), 187
mutation, 165

natural boundary condition, 363
nearer_the_eden(), 181, 182
neighboring cells, 162
Nelder–Mead simplex algorithm, 193
  ambiguities, 194
  centroid, 194, 202
  constrained, 211, 212, 247
  constrained to unconstrained, 212
  evaluation count, 197
  expand simplex, 195
  four cases, 194
  in one dimension, 194, 201
  inner contraction of simplex, 196
  length scale, 197, 200
  maxevals, 198
  outer contraction of simplex, 196
  problems/issues, 197
  reflect simplex, 194
  return value, 198
  shrink simplex, 196, 203
  stopping criterion, 196, 200, 204
  tolerance, 197
  unconstrained, 207
nelder_mead(), 204, 247
Netpbm library, 114
netpbm library and tools, 9
Neumann boundary condition, 302
Neumann boundary data, 361
Newton's iteration, 286
node (in a linked list), 147
nonlinear diffusion, 283
norm, 93
NUL character, *see also* ASCII NUL character, 66
  versus C's NULL pointer, 64
null terminator (in a string), 24

objective function, 193
  with parameters, 198, 200
orthogonal functions, 294
orthogonal matrix, 86, 98
orthogonal projection, 211, 213
orthonormal basis, 94, 95

PAM_STRUCT_SIZE, 120, 124
paraview, 8
PBM_FORMAT, 119
permutation matrix, 86
PGM_FORMAT, 119
pipe (Unix pipe), 16
pixel, 113
plain image storage format, 114
plainformat in *libnetpbm*, 119, 121

PlantEnergy, 164
plot_curve(), 275
plot_with_geomview_mono(), 329, 333, 350
plot_with_geomview_zhue(), 329, 333, 350
pm_close(), 119
pm_init(), 119
pm_openr(), 119
pm_openw(), 119
pnm_allocpamrow(), 120
pnm_freepamrow(), 120
pnm_readpaminit(), 120
pnm_readpamrow(), 120
pnm_writepaminit(), 120
pnm_writepamrow(), 120
pointer, 19
pointer arithmetic, 19
pointer to pointer, 56
pointer to void, 20
pointer vector, 56
Poisson equation, 337
poisson_solve(), 353, 370
population dynamics, *see* porous medium equation
porous medium equation, 283
    Barenblatt's solution, 283
    finite difference scheme, 285
    generalized, 284, 285
    population dynamics model, 289
Portable Bitmap (PBM), 73, 113, 114
Portable Graymap (PGM), 113, 116
Portable Pixmap (PPM), 114, 117
PPM_FORMAT, 119
print_animal(), 174
print_herd(), 174
print_matrix(), 60
print_vector(), 60
print_vector_loop_counter, 60
printf(), 15
process_link_line(), 238
process_load_line(), 238
process_node_line(), 235
projection operator, 211, 213
prune_matrix(), 140
putchar(), 15

qsort(), 21
quicksort, 155

rand(), 71
RAND_MAX, 71
random(), 72, 182, 184
random number generation, 71
rank-vertices-test.c, 201
rank_vertices(), 200
raw image storage format, 114
read_image(), 124
read_pgm_pixel_data(), 126
read_truss(), 231, 241, 247
read_wdf(), 169, 171
realloc(), 50, 69
redirection (Unix redirection), 16
reduce_pgm_image(), 142
reduce_ppm_image(), 142
reference configuration, 215
relative error, 137
remove_the_dead(), 182
replace_row(), 203
reproduce(), 187
reproduction, 164
ReproductionThreshold, 164
RGB color image, 117
rounding to nearest integer, 141
RPBM_FORMAT, 119
RPGM_FORMAT, 119
RPPM_FORMAT, 119

scanf(), 15
segmentation fault, 23
Seidman sweep, 260
    heat equation, 260
    porous medium equation, 286
        implementation, 286, 288
sentinel, 56, 57
shell, the Unix shell, 15
shrink(), 203
simplex, 193
    best vertex, 193
    degenerate, 193, 212
    next to worst vertex, 194
    program-generated, 199, 200
    user-prescribed, 199
    worst vertex, 194
Sobolev space, 340
solve_truss(), 230, 242, 244, 248
sparse_pack(), 81, 87
sparse_unpack(), 81
speciation, 162, 163
species, 161
square(), 316, 334, 350
square1(), 366
square2(), 366, 367
square3(), 366
square4(), 366, 369
square_f(), 334, 350
square_g(), 367
square_h(), 367
square_u_exact(), 350
srand(), 71
static declaration specifier
    for functions, 9

stderr, 15, 17
stdin, 15
stdout, 15, 17
stencil, 254
stored energy function, 219
stored_energy_function(), 242
strain, 219
stream, 15
    stderr, 15, 17
    stdin, 15
    stdout, 15, 17
stress, 218
stress_function(), 242, 243
stretch, 218
string joke, 24
string literal, 25
strings, 24
strlen(), 25, 70
strtod(), 27
strtof(), 28
strtol(), 28
strtold(), 28
strtoll(), 28
strtoul(), 28
strtoull(), 28
structure initialization, 6
support (of a function), 95, 284
symbolic link, 13
symlink, *see* symbolic link
system force vector, 341

Taylor, Wingate, and Bos (TWB) quadrature, 319
    strength, 321
    table sentinel, 324
three_holes(), 316, 335
three_holes_f(), 335, 350
total energy (of a truss), 222, 243
total_energy(), 243
transform(), 202
Triangle library, 8, 301, 313, 315
triangle_f(), 329, 349
triangle_with_hole(), 306, 315, 329, 331
triangular mesh, 301
triangulation, 301, 342
tridiagonal, 257, 260, 274
triplet form (in UMFPACK), 91–92, 353
trisolve(), 274, 280
truss, 215
    deformed configuration, 215
    evaluate_reactions(), 243, 247
    evaluate_stresses(), 242, 247
    free_links_list(), 237
    free_nodes_list(), 237
    free_truss(), 231, 239, 249
    geometric nonlinearity, 217
    get_links(), 232, 237
    get_loads(), 232, 238
    get_nodes(), 231, 234
    joint, 215
    link, 215
    link_stretch(), 242
    make_link(), 238
    make_node(), 236
    mechanical nonlinearity, 217
    member, 215
    minimum energy, 224
    node, 215
    node coordinates, 227, 245
    process_link_line(), 238
    process_load_line(), 238
    process_node_line(), 235
    read_truss(), 231, 241, 247
    reference configuration, 215
    rx and ry, 227, 237, 239, 243
    small deformation, 217
    solve_truss(), 230, 242, 244, 248
    solving, 216
    stored_energy_function(), 242
    stress, 239, 242
    stress_function(), 242, 243
    support, 216
    support reaction, 216, 239, 243
    topology design, 215
    total energy, 222, 243
    total_energy(), 243
    Truss Description File (TDF), 224
    truss_to_eps(), 240, 248
    write_truss(), 231, 239, 241, 248
    xfixed and yfixed, 227
truss_to_eps(), 240, 248
turn(), 185
twb_qdat(), 322, 331
TWB_STANDARD_AREA, 322

UMFPACK libary, 8, 85, 353
    numeric analysis, 86
    symbolic analysis, 85
    triplet form, 91–92, 353
umfpack-demo1.c, 87
umfpack-demo2.c, 89
umfpack-demo3.c, 90
umfpack_di_free_numeric(), 89
umfpack_di_free_symbolic(), 89
umfpack_di_numeric(), 86
umfpack_di_solve(), 86
umfpack_di_symbolic(), 85
umfpack_di_triplet_to_col(), 91, 353
UMFPACK_ERROR_out_of_memory, 89

UMFPACK_OK, 89
UMFPACK_WARNING_singular_matrix, 89
ungetc(), 15
update_world(), 187, 188

valgrind, 47
variable-length arrays, 23
void pointer, *see* pointer to void

wavelet decomposition, 100
wavelets in image analysis, 135
weak formulation of a BVP, 340, 362–363
whitespace, *see also* isspace(), 66
world, 162
    center cell, 162
    center cell formula, 163
    height and width, 166
    neighboring cells, 162
    torus (toroidal), 162, 186
World Definition File (WDF), 165
world_to_eps(), 175
write_image(), 126
write_pgm_pixel_data(), 128
write_truss(), 231, 239, 241, 248
write_wdf(), 174

xmalloc(), 50
xmalloc module, 45

Young's modulus, 220, 243