This volume contains Chapter 1 and the generic and C components of Chapters 9, 10, 11, 13, 14, and 15 from the second edition of the book *Computation and Problem Solving in Undergraduate Physics* (*CPSUP*). The first edition of *CPSUP* was registered with the Library of Congress with the call number QC20.C66.2004.

The second edition was registered with the Library of Congress with the LCCN 2019947884. It has been assigned the ISBN 9780961342968 and carries the Library of Congress call number QC20.C66.2019.
Preface

Note: Regardless of which components are included and which omitted in this version of Computation and Problem Solving in Undergraduate Physics, the acknowledgements, and disclaimer in the front matter, Appendix Z, and the end-of-chapter exercises are those from the assemblage containing all components. Some of those exercises may require use of tools not described in this version.

This volume contains Chapter 1, the generic and C components of Chapters 9, 10, 11, 13, 14, and 15, and Appendix Z from the second edition of the book Computation and Problem Solving in Undergraduate Physics (CPSUP). Chapter 1, the generic components in the identified chapters, and Appendix Z duplicate those components in other versions. This volume is self-contained and can be used alone, except that IDL, MATLAB, OCTAVE, and PYTHON are used to create graphical displays of solutions generated by C programs, and MAXIMA, MAPLE, and MATHEMATICA are used to evaluate a very few integrals arising in the description of finite element analysis for PDEs in Chapter 15.\(^1\) This chapter provides only a start on the included topics. References to more detailed descriptions of these resources are distributed throughout the text.

A few sections in Chapters 10, 11, 13, and 14 make use of C subroutines from the commercial package NUMERICAL RECIPES. A few sections in Chapter 15 exploit the ability of C programs to call FORTRAN routines, specifically routines from the package LAPACK. Very brief descriptions of these packages and information about how to acquire them are included in Appendix Z.

The much larger book CPSUP, which has been under continuous development and refinement at Lawrence University since the mid 1980’s, was first published in 2003 with a second edition somewhat expanded from the first edition coming available in 2017.\(^2\) CPSUP is a flexible, customizable text designed to

1. support efforts to acquaint students with computational procedures and resources early enough so that they will be motivated and prepared to use these resources on their own initiative when circumstances warrant and so that later work need not be interrupted to deal with computational issues as an aside to its main purposes, and

2. provide students with both the background and the confidence to support informed reading of vendor manuals, which usually do a splendid job of listing capabilities exhaustively but typically burden the beginner with initially irrelevant refinements and fail to illustrate adequately how even the rudimentary capabilities can be combined to perform useful tasks.

---

\(^1\)IDL, MATLAB, OCTAVE, PYTHON, MAXIMA, MAPLE, and MATHEMATICA are described in chapters that are omitted from this version of CPSUP but are included in other versions.

\(^2\)At its inception, CPSUP was self-published, since its substantial customizability to include or exclude discussion of several computational tools—IDL, MATLAB, OCTAVE, PYTHON, MAXIMA, MAPLE, MATHEMATICA, FORTRAN, C, Numerical Recipes, LSODE, MUDPACK, IBP\(\_\)X, TGIF, …—cannot even now be accommodated by commercial publishers. The article by David M. Cook titled “Computation in Undergraduate Physics: The Lawrence Approach” and appearing in the American Journal of Physics (Am. J. Phys. 76, 321–326 (April-May 2008)) describes these efforts in some detail. More information about CPSUP and the project that created it will remain for a time available at the website www.lawrence.edu/fast/cookd/ccli. Much of this information, however, is in the process of being moved to the Physical Sciences Resource Center of the AAPT ComPADRE Digital Library psrc.aapt.org/curricula/csup, a library that is part of the National Science Digital Library.
At Lawrence University, portions of CPSUP are used in a required sophomore course titled Computational Mechanics (CompMech) in which students develop an intermediate understanding of classical mechanics but also start the process of learning how to use available computational tools to pursue their studies in physics. The \LaTeX{} Appendix fosters an acquaintance with what may well be the most versatile tool for publishing scientific results, and students in that course are expected to submit some of their problem solutions as polished \LaTeX{} documents. The hope is that the materials treated in CPSUP and in CompMech will provide strong background so that instructors in subsequent courses can assign computational exercises and students can in subsequent courses have the confidence to exploit computational resources on their own initiative.

David M. Cook
Appleton, Wisconsin
13 February 2023
First and foremost, I wish to acknowledge the assistance and contributions of Peter Strunk, LU ’89, Kristi R. G. Hendrickson, LU ’91, Todd G. Ruskell, LU ’91, Stephen L. Mielke, LU ’92, Ruth Rhodes, LU ’92, Michelle Ruprecht, LU ’92, Sandra Collins, LU ’93, Mark F. Gehlke, LU ’93, Karl J. Geissler, LU ’94, Steven Van Metre, LU ’94, Alain Bellon, LU ’95, Peter Kelly Senecal, LU ’95, Christopher C. Schmidt, LU ’97, Michael D. Stenner, LU ’97, Mark Nornberg, LU ’98, Scot Shaw, LU ’98, Jim Truitt, LU ’98, Eric D. Moore, LU ’99, Teresa K. Hayne, LU ’00, Danica Dralus, LU, ’02, Ryan T. Peterson, LU ’03, Scott J. Kaminski, LU ’04, Michelle L. Milne, LU ’04, Lauren E. Kost, LU ’05, Claire Weiss, LU ’07, and Erik Garbackik, LU ’08—all Lawrence students who, as undergraduates, contributed during the first decade and a half to the evolution of this book and the accompanying manual of solutions to representative exercises. While current versions in most instances deviate—often considerably—from the initial drafts to which these students contributed, there can be no denying that their contributions have played an important role in the evolution of the use of computational resources in the Lawrence curriculum and the evolution of this book. I thank them warmly and sincerely for their assistance. All of these students have given permission for me to use whatever has over the years become of their contributions in this publication.

Second, I wish to acknowledge and thank several individuals who have provided reviews of drafts or otherwise assisted in the refinement of this book, including

- numerous students (beyond those specifically named in the previous paragraph) who have enrolled in my courses over the years and who, directly and indirectly, have made comments that have influenced this book.
- my departmental colleagues J. Bruce Brackenridge, John R. Brandenberger, Jeffrey A. Collett, Matthew R. Stoneking, and Paul W. Fontana, who have listened to my sometimes lengthy observations about matters computational and curricular and offered valuable criticisms.
- Wolfgang Christian (Davidson College), Robert Ehrlich (George Mason University), and A. John Mallinckrodt (California Polytechnic Institute–Pomona), each of whom participated in a 1990 Sloan-supported conference on computation in advanced undergraduate physics that I organized at Lawrence and each of whom as a consultant throughout the writing project has offered many suggestions and constructive criticisms.
- all those faculty members¹ who participated in one of the NSF-supported week-long workshops offered at Lawrence in the summers of 2001, 2002, and 2003 and who, in that context,

¹In July 2001: Steve Adams ( Widener University), Russel Kauffman (Muhlenberg College), Marylin Bell (Lake Forest College), Lawrence A. Molnar (Calvin College), Thomas R. Greenlee (Bethel College), Elliott Moore (New Mexico Tech), Javier Hasbun (State University of West Georgia), Joelle L. Murray (Linfield College), Derrick Hylton (Spelman College), Paula C. Turner (Kenyon College), Ross Hyman (DePaul University), Scott N. Walck (Lebanon Valley College), William H. Ingham (James Madison University), and Tim Young (University of North Dakota). In July, 2002 (first workshop): Albert Batten (United States Air Force Academy), Andrea Cox LU ’91 (Beloit College), Brian Cudnik (Prairie View A and M), Craig Gunsul (Whitman College), Kevin Lee (University of Nebraska), Kao-Biu Luk (University of California, Berkeley), Daryl Macon (Boise State University), Viktor Martisovits (Central College), Donald Miller (Central Missouri State University), Dorn Peterson (James Madison University), Robert Ragan (University of Wisconsin – Lacrosse), Shafiq Rahman (Allegheny College), Rahmatullah Syed (Norwich University), Jorge Talamantes (California State University – Bakersfield), Mark Timko (Elmhurst College), Paul Tjossem
ACKNOWLEDGEMENTS

provided a critical examination of the evolving manuscript and offered numerous suggestions for improvement—in addition to finding many previously undetected typographical glitches.

- especially Javier Hasbun (State University of West Georgia)—who had students working with almost the entire manuscript—but also William H. Ingham (James Madison University), Russel Kauffman (Muhlenberg College), Joelle L. Murray (Linfield College), Shafiq Rahman (Allegheny College), John Thompson (DePaul University), and Tim Young (University of North Dakota), all of whom used at least portions of the growing manuscript in their teaching during the academic year 2001-02 and who collectively sent me a wide spectrum of varied and valuable comments.

- several anonymous reviewers engaged by potential publishers as they evaluated my efforts, even though all potential publishers ultimately decided they could not provide the microscopic customization the book required.

Third, I wish to acknowledge considerable debt to many individuals whom I do not know but whose contributions behind the scenes have been invaluable. Chief among these individuals are

- Donald Knuth, originator of \TeX;
- Leslie Lamport, originator of \LaTeX;
- Han The Than, originator of pdf\TeX, which provides the engine underlying pdflatex for creating PDF files directly from \LaTeX source code.;
- Pehong Chen and Nelson Beebe, originator and current maintainer of makeindex for preparing indices to \LaTeX documents;
- Enrico Gregorio, originator and current maintainer of \imakeidx that reduces the number of passes required to format and include an index;
- David Carlisle and Sebastian Rahtz, originators, and David Carlisle, current maintainer, of the \LaTeX package \texttt{graphicx} for including graphic images in \LaTeX documents;
- Leslie Lamport, David Carlisle and other members of the \LaTeX3 team, authors of the \LaTeX package \texttt{ifthen} for supporting conditional statements in \LaTeX source files;
- Sebastian Rahtz and Heiko Oberdiek, originator of the \LaTeX package \texttt{hyperref} for creating linked versions of documents as PDF files;
- Paul Vojta, author of \texttt{xdvi}, a versatile on-screen previewer for the .dvi files produced by \TeX and \LaTeX;
- Till Tantau et. al., originators and current maintainers of the \LaTeX packages \texttt{tikz} and \texttt{pgf} for creating sophisticated diagrams using \LaTeX-like inclusions in the source file;
- William Chia-Wei Cheng, author of TGIF, a program used to create several of the figures;
- John Bradley, author of \texttt{xv}, a program used to convert a few bit-mapped files into PostScript;

(Grinnell College), and John Walkup (California Polytechnic Institute – San Luis Obispo). In July, 2002 (second workshop): Julio Blanco (California State University–Northridge), William Briscoe (George Washington University), Paul Bunson (Lawrence University), Albert Chen (Oklahoma Baptist University), K. Kelvin Cheng (Texas Tech University), Bob Davis (Taylor University), Mirela Petaa (University of Richmond), David Grumbine (St. Vincent College), Kristi Hendrickson LU ’91 (University of Puget Sound), Tom Herrmann (Eastern Oregon University), Seamus Lagan (Whittier College), Eric Lane (University of Tennessee at Chattanooga), Ernest Ma (Montclair State University), Norris Preyer (College of Charleston), Michael Schillaci (Francis Marion University), Mihir Sejpal (Earlham College), Laura Van Wormer (Hiram College), Mary Lou West (Montclair State University), and Douglas Young (Mercer University). In June, 2003: Stephen Addison (University of Central Arkansas), Mohan Aggarwal (Alabama A&M), Anne Caraley (SUNY at Oswego), William Dieterle (California University of Pennsylvania), Timothy Duman (University of Indianapolis), Brett Fadem (Colby College), Michael Gray (American University), James Meyer (St. Gregory’s University), Prabasaj Paul (Denison University), David Peterson (Francis Marion University), Edward Pogozelski (SUNY at Geneseo), Chandra Prayaga (University of West Florida), Bruce Richards (Oberlin College), David Seely (Albion College), Adam Smith (Hillsdale College), Walther Spjeldvik (Weber State University), Julie Talbot (State University of West Georgia), Paul Thomas (University of Wisconsin–Eau Claire), Srinivasa Venugopalan (SUNY at Binghamton), and Alma Zook (Pomona College).
• The many contributors to the MiKTeX project;
• The developers, authors, and maintainers of WinEdt, the text editor used during the latter years of writing CPSUP; and
• The developers and maintainers of ps2pdf for converting PostScript files to PDF and the developers and maintainers of pdfcrop for pruning excessive white borders from PDF files.
• Radical Eye Software, which holds the copyright on dvips, a program for converting .dvi files to PostScript.

Quite simply, this project would have been impossible without the availability of these several programs and utilities, each of which played a necessary role behind the scenes in preparing or processing the files from which, ultimately, a printable PostScript or PDF file for the finished book emerged.

Fourth, I point out that the names of several pieces of commercial software are, in fact, trademarks or registered trademarks belonging to the vendors of those software products. Each such trademark is identified at its first occurrence in the text proper, and detailed contact information for every vendor is compiled in Appendix Z.

Fifth, I acknowledge the following specific permissions, each of which is more fully explained at the point in the text where the permission is explicitly invoked. In particular, I thank

• The MathWorks, Inc., for permission to incorporate in this book and distribute IDL source code for the routines ludiffeq_23 and ludiffeq_45, which code uses algorithms patterned after those used in 1991 in the MATLAB routines ode23 and ode45.
• Wayne Landsman, author of the IDL routines qsimpson and trapzd in the IDL Astronomy User’s Library, for permission to use those routines as the basis for the routines luqsimp and lutrapzd and to distribute the source code for luqsimp and lutrapzd as supplements to this book.
• Research Systems (later Exelis Visual Information Solutions and now part of Harris Geospatial Solutions), Incorporated, for permission to use portions of any RSI-supplied and/or edited .pro code—most particularly evident in RSI contributions to ludiffeq_23.pro, ludiffeq_45.pro, and luqsimp.pro—and to use the IDL name and trademark.
• Numerical Recipes Software (a) for permission to use the names and calling sequences of several Numerical Recipes routines at various places in this book, (b) for permission to refer to the C header files nr.h and nrutil.h and the file nrutil.c containing assorted utilities used by various C recipes, and (3) for permission to use the names and calling sequences of several IDL routines that are derived from Numerical Recipes routines (and for the use of which Research Systems Incorporated has permission from Numerical Recipes Software).
• William Chia-Wei Cheng, author of TGIF, for permission to reproduce in the appendix on that program several of the icons used in its many screen displays.

Sixth, I wish to acknowledge and thank the W. M. Keck Foundation for three grants [#880969 (1988), #031248 (1993), #011795 (2001)], the National Science Foundation for three grants [USE-8851685 (ILI–1988), DUE-9350667 (ILI–1993), and DUE-9952285 (CCLI–EMD-2000)], and Lawrence University for matching and other support. 3 A small conference held at Lawrence in 1990 and supported by the Alfred P. Sloan Foundation stimulated thinking among the participants about

3Specifically flmoon, xflmoon, caldat, julday, xjulday, avevar, xavevar, rk4, xrk4, rkqs, rkck, mmid, bstep, rkdumb, odeint, trapzd, xtrapzd, qtrap, xqtrap, qsimp, qromb, point, rtbis, rtbisl, rntnewt, rntnewt, rtnewt, rtsafe, xtsafe, zbrak, gaussj, ludcmp, lubksb, tridag, svdcmp, svbsv, mnewt, newt, and brydn (both in FORTRAN and in C).
3Any opinions, findings, and conclusions or recommendations expressed in this book are those of the author and do not necessarily reflect the views of any of these granting foundations or agencies.
uses of computers in upper-division undergraduate physics. All of these grants have contributed in many ways to the developments at Lawrence that have culminated in the writing of this book. In particular, the NSF CCLI-EMD grant made in February, 2000, supported my sabbatical while I finalized the text of (the first edition of) this book. That grant also supported four week-long summer faculty workshops that have, on the one hand, provided constructive feedback on a succession of drafts and, on the other hand, enhanced awareness nationally of this book and of the developments at Lawrence.

Finally (and most important of all), I wish to express deep gratitude for the decades during which my wife, Cynthia, has shown patience, understanding, and support beyond any possibility of my ever fully knowing. In truth, I do not know how to thank her.
Disclaimer

The statements described in the various chapters of this book have been tested extensively but have certainly not been tested with all versions of all software packages on all possible platforms with all possible versions of the underlying operating systems. Differences from version to version of the software packages, from operating system to operating system, and from platform to platform exist. This brief section identifies the versions of the various programs that have been tested and the operating systems and platforms on which those tests have been carried out. That the behavior of other combinations of version, operating system, and platform will conform in every detail to that herein described can, of course, not be guaranteed. One can, however, have some confidence that the behavior in combinations not explicitly tested will not differ enormously from that described herein—except that newer versions of a software package may well have features not implemented in earlier versions (and occasionally a feature or specific syntax available in an earlier version has been removed altogether from more recent versions). With reasonable confidence, one can presume that the commands and syntax and features described in this book will work on other platforms with the tested versions of the programs and with subsequent versions. Statements herein that exploit features implemented for the first time in the tested versions will, of course, not be accepted in earlier versions, but those “glitches” should not be numerous or extensive. Where, in the months and years since the original draft was created, I have become aware of such glitches, I have made appropriate updates and subsequent productions have incorporated those updates. Nothing, however, assures that I have identified all such glitches.

That disclaimer having been stated, I now present for each program a brief tally of the version(s) tested and the platform(s) and operating system(s) on which those tests have been carried out:

- The MAXIMA codings herein have been fully verified with
  - MAXIMA Version 5.36.1 and wxMAXIMA Version 15.04.0 on a Hewlett-Packard platform running Windows 7,
  - MAXIMA Version 5.38.1 and wxMAXIMA Version 16.04.2 on a Hewlett-Packard platform running Windows 7, and
  - MAXIMA Version 5.36.1 and wxMAXIMA Version 12.01.0 on a Hewlett-Packard platform running the Fedora 17 implementation of LINUX.

  In addition, these codings have been spot-checked with
  - MAXIMA Version 5.38.0 and wxMAXIMA 16.04.1,
  - MAXIMA 5.39.0 and wxMAXIMA 16.12.0.

1 The date of production of each version of CPSUP is displayed at the top of the cover page on that version. I have maintained a dated list of edits made to the source files, so changes made after the date of production of a particular version of CPSUP and a subsequent production of that version can readily be identified for anyone who wishes to update an outdated production. Generally, updated productions fairly promptly replace the previous production at parc.aapt.org/curricula/cpsup. Versions dated between 10 and 31 January 2021 provide the base. Edits made after 31 January 2021 are recorded in the file of edits.
DISCLAIMER

- MAXIMA 5.42.0 and wxMAXIMA 18.10.1 on a Hewlett-Packard platform running Windows 10.

- The MAPLE codings herein have been fully verified with MAPLE Version 16 on a Hewlett-Packard platform running Windows 7 and on a Hewlett-Packard platform running the Fedora 17 implementation of LINUX. These codings have also been spot-checked with MAPLE Version 17 on a Hewlett-Packard platform running Windows 10.

- The Mathematica codings have been fully verified with Mathematica Version 11.3 on a Hewlett-Packard platform running Windows 7 and on a Hewlett-Packard platform running the Fedora 17 implementation of LINUX. The Mathematica codings have also been spot-checked with Mathematica 12.0 on a Hewlett-Packard platform running Windows 10.

- The IDL details codings have been fully verified with IDL Versions 8.3 and 8.5 on a Hewlett-Packard platform running Windows 7 and a Hewlett-Packard platform running the Fedora 17 implementation of LINUX. These versions of IDL have also been spot-checked on a Hewlett-Packard platform running Windows 10.

- The MATLAB codings have been fully verified with MATLAB Version R2012a on a Hewlett-Packard Platform running Windows 7 and a Hewlett-Packard platform running the Fedora 17 implementation of LINUX. These codings have also been spot-checked on a Hewlett-Packard platform running Windows 10.

- The OCTAVE codings have been fully verified with
  - OCTAVE Version 4.0.0 on a Hewlett-Packard platform running Windows 7,
  - OCTAVE Version 3.6.3 on a Hewlett-Packard platform running the Fedora 17 implementation of LINUX, and
  - OCTAVE Version 4.0.3 on a Hewlett-Packard platform running the Fedora 25 implementation of LINUX.

OCTAVE Version 4.2.2 has been spot-checked on a Hewlett-Packard platform running Windows 7, and OCTAVE Versions 4.0.0 and 5.2.0 have been spot-checked on a Hewlett-Packard platform running Windows 10.

- Except where otherwise noted in the text, the PYTHON codings have been fully verified with
  - PYTHON 2.7.16 from the Anaconda2 distribution installed on a Hewlett-Packard platform running Windows 10, using the Anaconda2 prompt and also using the Anaconda2 Python Shell,
  - PYTHON 3.7.3 from the Anaconda3 distribution installed on a Hewlett-Packard platform running Windows 10, using the Anaconda3 prompt and also using the Anaconda3 Python Shell.

The codings in the PYTHON chapter and some of the PYTHON codings in other chapters have been verified with PYTHON 3.9.13 from the Anaconda3 distribution installed on a Hewlett-Packard platform running Windows 10, using the Anaconda3 Python shell.

- The Numerical Recipes codings have been fully verified with Numerical Recipes Version 2.10 only on a Hewlett-Packard platform running the Fedora 17 implementation of LINUX. Those codings have also been spot-checked on a Hewlett-Packard platform running Windows 10 using 64-bit GNU Fortran Version 7.1.0 and 64-bit GNU C Version 7.1.0.

\[ ^2 \text{See the Local Guide for ways to bring up the prompt and the shell in your environment.} \]

\[ ^3 \text{See the previous footnote.} \]
• The LSODE codings have been fully verified with LSODE whose README file bears the date 30 March 1987 only on a Hewlett-Packard platform running the Fedora-17 implementation of LINUX. The codings have also been spot-checked with LSODE whose README file (opkd-sum) bears the date 20 June 2001 on a Hewlett-Packard platform running Windows 10 with 64-bit GNU Fortran Version 7.1.0.

• The \LaTeX{} details apply specifically to \LaTeX{} 2ε with the MiK\TeX{} implementation on a Hewlett-Packard platform running Windows 7 and on a Hewlett-Packard platform running Windows 10. \LaTeX{} normally responds to the same source code on all platforms.

• The TGIF details apply specifically to TGIF Version 4.2 (patchlevel 5) on a Hewlett-Packard platform running the Fedora 17 implementation of LINUX. (TGIF is exclusively a UNIX package.) Other versions and patchlevels and other platforms will surely have similar behavior, but may not conform exactly to the behavior here described.

• The MUDPACK codings have been fully verified with MUDPACK Version 5.0.1 whose README file bears the date 6 December 2011 only on a Hewlett-Packard platform running Windows 10 with 64-bit GNU Fortran Version 7.1.0.
# Table of Contents

## Preface

- iii

## Acknowledgements

- v

## Disclaimer

- ix

## Table of Contents

- xiii

### 1 Preliminaries

1.1 An Orientation to Computers ........................................ 2
   - 1.1.1 A Simple Responsive Machine ................................ 3
   - 1.1.2 Character Codes ........................................... 4
   - 1.1.3 The ASCII Character Set .................................. 5
   - 1.1.4 Representation of Data in a Computer ....................... 6
1.2 Files and Directories ............................................. 8
1.3 Operating Systems ................................................. 9
1.4 Glossary, Conventions, and Understandings ...................... 9
1.5 Assumed Background ............................................... 11
   - 1.5.1 The Gamma Function ...................................... 12
   - 1.5.2 The Laplace Transform .................................... 13
1.6 Licensing Issues .................................................. 14

### 9 Introduction to Programming

9.1 Components of a Programming Language ......................... 16
   - 9.1.1 Variables, Variable Names, and Use of Memory .......... 16
   - 9.1.2 Essential Action Statements ............................. 18
   - 9.1.3 Logical Expressions and Conditions ..................... 19
   - 9.1.4 Syntactic Wrinkles ...................................... 20
   - 9.1.5 Essential Control Structures ............................ 20
9.2 Sample Short Algorithms ......................................... 24
9.3 Two Larger Algorithms .......................................... 26
   - 9.3.1 Solving Laplace’s Equation .............................. 26
   - 9.3.2 File Output/Input ........................................ 28
9.6 Solving Laplace’s Equation with C ................................ 31
9.7 Additional Features of C ....................................... 36
   - 9.7.1 Multi-Way Selection ..................................... 36
   - 9.7.2 Reading Data from Keyboard .............................. 36
   - 9.7.3 Additional Loops ......................................... 37
   - 9.7.4 Functions and Global Variables .......................... 37
9.12 Creating and Storing Two-Dimensional Scalar Arrays ............ 38
   - 9.12.2 ... with C ............................................. 39
9.13 Creating and Storing Three-Dimensional Scalar Arrays ........... 41
   - 9.13.2 ... with C ............................................. 41
9.14 Creating and Storing Two-Dimensional Vector Arrays ............. 42
   - 9.14.2 ... with C ............................................. 44
9.15 Creating and Storing Three-Dimensional Vector Arrays .......... 44
   - 9.15.2 ... with C ............................................. 46

- xiii
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.16</td>
<td>Reading Files</td>
<td>47</td>
</tr>
<tr>
<td>9.16.2</td>
<td>... with C</td>
<td>47</td>
</tr>
<tr>
<td>9.17</td>
<td>References</td>
<td>48</td>
</tr>
<tr>
<td>9.18</td>
<td>Exercises</td>
<td>48</td>
</tr>
<tr>
<td>9.B</td>
<td>Listing of <code>irrad.c</code></td>
<td>55</td>
</tr>
<tr>
<td>9.I</td>
<td>Listing of <code>pdens.c</code></td>
<td>57</td>
</tr>
<tr>
<td>9.L</td>
<td>Listing of <code>wavegd.c</code></td>
<td>59</td>
</tr>
<tr>
<td>9.O</td>
<td>Listing of <code>quadpole.c</code></td>
<td>61</td>
</tr>
<tr>
<td>9.U</td>
<td>Listing of <code>read_irrad.c</code></td>
<td>63</td>
</tr>
<tr>
<td>10</td>
<td>Introduction to Numerical Recipes</td>
<td>65</td>
</tr>
<tr>
<td>10.1</td>
<td>The Numerical Recipes Directory Tree</td>
<td>65</td>
</tr>
<tr>
<td>10.3</td>
<td>Using C Recipes</td>
<td>66</td>
</tr>
<tr>
<td>10.4</td>
<td>Exercises</td>
<td>68</td>
</tr>
<tr>
<td>10.5</td>
<td>Using Numerical Recipes</td>
<td>69</td>
</tr>
<tr>
<td>11</td>
<td>Solving Ordinary Differential Equations</td>
<td>69</td>
</tr>
<tr>
<td>11.1</td>
<td>Sample Problems</td>
<td>70</td>
</tr>
<tr>
<td>11.1.1</td>
<td>Projectile in a Viscous Medium</td>
<td>70</td>
</tr>
<tr>
<td>11.1.2</td>
<td>Chain Radioactive Decay</td>
<td>73</td>
</tr>
<tr>
<td>11.1.3</td>
<td>Exponential and Logistic Growth</td>
<td>74</td>
</tr>
<tr>
<td>11.1.4</td>
<td>Forced, Driven, Damped Harmonic Oscillation</td>
<td>74</td>
</tr>
<tr>
<td>11.1.5</td>
<td>An LRC Resonant Circuit</td>
<td>76</td>
</tr>
<tr>
<td>11.1.6</td>
<td>Coupled Oscillators</td>
<td>76</td>
</tr>
<tr>
<td>11.1.7</td>
<td>Motion under Central Forces</td>
<td>77</td>
</tr>
<tr>
<td>11.1.8</td>
<td>Standing Waves in a String</td>
<td>79</td>
</tr>
<tr>
<td>11.1.9</td>
<td>The Schrödinger Equation in One Dimension</td>
<td>80</td>
</tr>
<tr>
<td>11.2</td>
<td>Laplace Transforms</td>
<td>81</td>
</tr>
<tr>
<td>11.6</td>
<td>Algorithms for Solving ODEs Numerically</td>
<td>82</td>
</tr>
<tr>
<td>11.6.1</td>
<td>Euler's Method</td>
<td>82</td>
</tr>
<tr>
<td>11.6.2</td>
<td>Improved Euler Method</td>
<td>84</td>
</tr>
<tr>
<td>11.6.3</td>
<td>Runge-Kutta Methods</td>
<td>85</td>
</tr>
<tr>
<td>11.6.4</td>
<td>Assessing Accuracy</td>
<td>88</td>
</tr>
<tr>
<td>11.6.5</td>
<td>Adaptive Methods</td>
<td>90</td>
</tr>
<tr>
<td>11.6.6</td>
<td>Multistep Methods</td>
<td>90</td>
</tr>
<tr>
<td>11.15</td>
<td>Solving ODEs Numerically with C</td>
<td>91</td>
</tr>
<tr>
<td>11.15.1</td>
<td>C Programs from Scratch</td>
<td>91</td>
</tr>
<tr>
<td>11.15.2</td>
<td>Using Numerical Recipes</td>
<td>93</td>
</tr>
<tr>
<td>11.16</td>
<td>Exercises</td>
<td>98</td>
</tr>
<tr>
<td>11.16.1</td>
<td>... using Symbolic Methods</td>
<td>98</td>
</tr>
<tr>
<td>11.16.2</td>
<td>... using Numerical Methods</td>
<td>102</td>
</tr>
<tr>
<td>11.16.3</td>
<td>... using Numerical Recipes</td>
<td>105</td>
</tr>
<tr>
<td>11.D</td>
<td>Listing of <code>decay.c</code></td>
<td>107</td>
</tr>
<tr>
<td>11.E</td>
<td>Listing of <code>decarkr4.c</code></td>
<td>108</td>
</tr>
<tr>
<td>11.F</td>
<td>Listing of <code>decarkrkqs.c</code></td>
<td>109</td>
</tr>
<tr>
<td>13</td>
<td>Evaluating Integrals</td>
<td>111</td>
</tr>
<tr>
<td>13.1</td>
<td>Sample Problems</td>
<td>111</td>
</tr>
<tr>
<td>13.1.1</td>
<td>One-Dimensional Trajectories</td>
<td>111</td>
</tr>
<tr>
<td>13.1.2</td>
<td>Center of Mass</td>
<td>112</td>
</tr>
<tr>
<td>13.1.3</td>
<td>Moment of Inertia; Radius of Gyration</td>
<td>113</td>
</tr>
<tr>
<td>13.1.4</td>
<td>The Large Amplitude Simple Pendulum</td>
<td>114</td>
</tr>
<tr>
<td>13.1.5</td>
<td>Statistical Data Analysis</td>
<td>115</td>
</tr>
<tr>
<td>13.1.6</td>
<td>The Corn Spiral</td>
<td>116</td>
</tr>
<tr>
<td>13.1.7</td>
<td>Electric and Magnetic Fields and Potentials</td>
<td>116</td>
</tr>
<tr>
<td>13.1.8</td>
<td>Quantum Probabilities</td>
<td>118</td>
</tr>
<tr>
<td>13.1.9</td>
<td>Expansion in Orthogonal Functions</td>
<td>118</td>
</tr>
<tr>
<td>13.5</td>
<td>Algorithms for Numerical Integration</td>
<td>119</td>
</tr>
<tr>
<td>13.5.1</td>
<td>Newton-Cotes Quadrature</td>
<td>119</td>
</tr>
<tr>
<td>13.5.2</td>
<td>Rearrangements for Computational Efficiency</td>
<td>121</td>
</tr>
<tr>
<td>13.5.3</td>
<td>Assessing Error</td>
<td>123</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>13.5.4</td>
<td>Iterative and Adaptive Algorithms</td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>13.5.5</td>
<td>Gaussian Quadrature</td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>13.14</td>
<td>Evaluating Integrals Numerically with C</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>13.14.1</td>
<td>Writing Programs from Scratch</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>13.14.2</td>
<td>Using Numerical Recipes</td>
<td>130</td>
<td></td>
</tr>
<tr>
<td>13.15</td>
<td>Exercises</td>
<td>133</td>
<td></td>
</tr>
<tr>
<td>13.15.1</td>
<td>using Symbolic Methods</td>
<td>133</td>
<td></td>
</tr>
<tr>
<td>13.15.2</td>
<td>using Numerical Methods</td>
<td>136</td>
<td></td>
</tr>
<tr>
<td>13.15.3</td>
<td>using Numerical Recipes</td>
<td>140</td>
<td></td>
</tr>
<tr>
<td>13.17</td>
<td>Exercises</td>
<td>166</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>Finding Roots</td>
<td>143</td>
<td></td>
</tr>
<tr>
<td>14.1</td>
<td>Sample Problems</td>
<td>143</td>
<td></td>
</tr>
<tr>
<td>14.1.1</td>
<td>Classical Turning Points</td>
<td>144</td>
<td></td>
</tr>
<tr>
<td>14.1.2</td>
<td>A Max-Min Problem: Equilibrium Points</td>
<td>144</td>
<td></td>
</tr>
<tr>
<td>14.1.3</td>
<td>Natural Frequencies of Oscillating Systems</td>
<td>145</td>
<td></td>
</tr>
<tr>
<td>14.1.4</td>
<td>Range of Projectile in Viscous Medium</td>
<td>146</td>
<td></td>
</tr>
<tr>
<td>14.1.5</td>
<td>Energy Levels in a Quantum Well</td>
<td>149</td>
<td></td>
</tr>
<tr>
<td>14.2</td>
<td>Symbolic Approaches</td>
<td>150</td>
<td></td>
</tr>
<tr>
<td>14.6</td>
<td>Algorithms for Finding Roots Numerically</td>
<td>151</td>
<td></td>
</tr>
<tr>
<td>14.6.1</td>
<td>The Method of Bisection</td>
<td>151</td>
<td></td>
</tr>
<tr>
<td>14.6.2</td>
<td>Newton’s Method</td>
<td>152</td>
<td></td>
</tr>
<tr>
<td>14.6.3</td>
<td>Other Methods</td>
<td>152</td>
<td></td>
</tr>
<tr>
<td>14.6.4</td>
<td>Assessing Accuracy</td>
<td>153</td>
<td></td>
</tr>
<tr>
<td>14.15</td>
<td>Finding Roots Numerically with C</td>
<td>154</td>
<td></td>
</tr>
<tr>
<td>14.15.1</td>
<td>C Programs from Scratch</td>
<td>154</td>
<td></td>
</tr>
<tr>
<td>14.15.2</td>
<td>Using Numerical Recipes</td>
<td>157</td>
<td></td>
</tr>
<tr>
<td>14.16</td>
<td>Solving Simultaneous Equations</td>
<td>161</td>
<td></td>
</tr>
<tr>
<td>14.16.1</td>
<td>Systems of Linear Equations</td>
<td>161</td>
<td></td>
</tr>
<tr>
<td>14.16.2</td>
<td>Systems of Nonlinear Equations</td>
<td>164</td>
<td></td>
</tr>
<tr>
<td>14.17</td>
<td>Exercises</td>
<td>166</td>
<td></td>
</tr>
<tr>
<td>14.17.1</td>
<td>using Symbolic Methods</td>
<td>166</td>
<td></td>
</tr>
<tr>
<td>14.17.2</td>
<td>using Numerical Methods</td>
<td>168</td>
<td></td>
</tr>
<tr>
<td>14.17.3</td>
<td>using Numerical Recipes</td>
<td>171</td>
<td></td>
</tr>
<tr>
<td>14.17.4</td>
<td>Finding More than One Unknown</td>
<td>171</td>
<td></td>
</tr>
<tr>
<td>14.17E</td>
<td>Listing of <code>trapezoidal.c</code></td>
<td>176</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Solving Partial Differential Equations</td>
<td>177</td>
<td></td>
</tr>
<tr>
<td>15.1</td>
<td>Sample Problems</td>
<td>179</td>
<td></td>
</tr>
<tr>
<td>15.1.1</td>
<td>Motion of a String: The Wave Equation</td>
<td>179</td>
<td></td>
</tr>
<tr>
<td>15.1.2</td>
<td>Heat Flow: The Diffusion Equation</td>
<td>182</td>
<td></td>
</tr>
<tr>
<td>15.1.3</td>
<td>Steady State Heat Flow: The Laplace Equation</td>
<td>183</td>
<td></td>
</tr>
<tr>
<td>15.1.4</td>
<td>Other Situations</td>
<td>183</td>
<td></td>
</tr>
<tr>
<td>15.1.5</td>
<td>Classification of Second-Order PDEs</td>
<td>184</td>
<td></td>
</tr>
<tr>
<td>15.1.6</td>
<td>Equations of Fluid Dynamics: Navier-Stokes’ Equations</td>
<td>185</td>
<td></td>
</tr>
<tr>
<td>15.1.6.1</td>
<td>Conservation of Mass: The Equation of Continuity</td>
<td>186</td>
<td></td>
</tr>
<tr>
<td>15.1.6.2</td>
<td>Newton’s Second Law</td>
<td>187</td>
<td></td>
</tr>
<tr>
<td>15.1.6.3</td>
<td>A Special Case: Non-viscous Flow</td>
<td>189</td>
<td></td>
</tr>
<tr>
<td>15.1.6.4</td>
<td>A Second Special Case: Sound Waves</td>
<td>190</td>
<td></td>
</tr>
<tr>
<td>15.1.7</td>
<td>A General 1D Equation</td>
<td>191</td>
<td></td>
</tr>
<tr>
<td>15.1.8</td>
<td>A General 2D Equation</td>
<td>192</td>
<td></td>
</tr>
<tr>
<td>15.2</td>
<td>Finite Difference Methods (FDMs) in One Dimension</td>
<td>192</td>
<td></td>
</tr>
<tr>
<td>15.8</td>
<td>Using C to Solve 1D PDEs via an FDM</td>
<td>195</td>
<td></td>
</tr>
<tr>
<td>15.8.1</td>
<td>A General Coding using LAPACK</td>
<td>195</td>
<td></td>
</tr>
<tr>
<td>15.8.2</td>
<td>An Example: Simple Harmonic Motion</td>
<td>198</td>
<td></td>
</tr>
<tr>
<td>15.8.2.1</td>
<td>Setting the Problem</td>
<td>198</td>
<td></td>
</tr>
<tr>
<td>15.8.2.2</td>
<td>Running the C Program</td>
<td>198</td>
<td></td>
</tr>
<tr>
<td>15.8.2.3</td>
<td>Displaying the Solution Graphically with IDL</td>
<td>199</td>
<td></td>
</tr>
<tr>
<td>15.8.2.4</td>
<td>Displaying the Solution Graphically with MATLAB</td>
<td>200</td>
<td></td>
</tr>
<tr>
<td>15.8.2.5</td>
<td>Displaying the Solution Graphically with OCTAVE</td>
<td>200</td>
<td></td>
</tr>
</tbody>
</table>
15.8.2.6 Displaying the Solution Graphically with PYTHON .......................... 201
15.9 Finite Element Methods (FEMs) in One Dimension ............................. 202
15.9.1 Discretizing the Domain: Preprocessing ............................. 203
15.9.2 Selecting Interpolation or Shape Functions ............................. 203
15.9.3 Formulating the Equations for a Single Element ......................... 205
15.9.4 Assembling the System of Equations ............................. 207
15.9.5 Incorporating the Boundary Conditions ............................. 210
15.15 Using C to Solve 1D PDEs via an FEM ........................................... 211
15.15.1 A General Coding using LAPACK ........................................... 211
15.15.2 An Example: Simple Harmonic Motion ........................................... 214
15.15.2.1 Setting the Problem ........................................... 214
15.15.2.2 Running the C Program ........................................... 215
15.15.2.3 Displaying the Solution Graphically with IDL ........................................... 215
15.15.2.4 Displaying the Solution Graphically with MATLAB ........................................... 216
15.15.2.5 Displaying the Solution Graphically with OCTAVE ........................................... 217
15.15.2.6 Displaying the Solution Graphically with PYTHON ........................................... 218
15.16 Finite Difference Methods (FDMs) in Two Dimensions ............................. 219
15.16.1 The Wave Equation ........................................... 219
15.16.2 The Diffusion Equation ........................................... 222
15.16.3 The Laplace Equation ........................................... 224
15.22 Using C to Solve 2D PDEs via an FDM ........................................... 227
15.22.1 The Wave Equation ........................................... 227
15.22.1.1 The C Coding ........................................... 227
15.22.1.2 Running the C Program ........................................... 231
15.22.1.3 Displaying the Solution Graphically with IDL ........................................... 231
15.22.1.4 Displaying the Solution Graphically with MATLAB ........................................... 233
15.22.1.5 Displaying the Solution Graphically with OCTAVE ........................................... 234
15.22.1.6 Displaying the Solution Graphically with PYTHON ........................................... 236
15.22.2 The Diffusion Equation ........................................... 238
15.22.2.1 The C Coding ........................................... 238
15.22.2.2 Running the C Program ........................................... 241
15.22.2.3 Displaying the Solution Graphically with IDL ........................................... 241
15.22.2.4 Displaying the Solution Graphically with MATLAB ........................................... 243
15.22.2.5 Displaying the Solution Graphically with OCTAVE ........................................... 244
15.22.2.6 Displaying the Solution Graphically with PYTHON ........................................... 246
15.22.3 The Laplace Equation ........................................... 248
15.22.3.1 Creating and Testing the C Coding ........................................... 248
15.22.3.2 Displaying the Solution Graphically with IDL ........................................... 253
15.22.3.3 Displaying the Solution Graphically with MATLAB ........................................... 254
15.22.3.4 Displaying the Solution Graphically with OCTAVE ........................................... 256
15.22.3.5 Displaying the Solution Graphically with PYTHON ........................................... 258
15.23 Finite Element Methods (FEMs) in Two Dimensions ............................. 260
15.23.1 Discretizing the Domain: Preprocessing ........................................... 260
15.23.2 Selecting the Interpolation or Shape Functions ........................................... 262
15.23.3 Formulating the Equations for a Single Element ........................................... 263
15.23.4 Assembling the System of Equations ........................................... 265
15.23.5 Incorporating the Boundary Conditions ........................................... 267
15.29 Using C to Solve 2D PDEs via an FEM ........................................... 269
15.29.1 A General Coding using LAPACK ........................................... 269
15.29.2 An Example: Isotropic Heat Flow ........................................... 278
15.29.2.1 Setting the Problem ........................................... 278
15.29.2.2 Running the C Program ........................................... 278
15.29.2.3 Displaying the Solution Graphically with IDL ........................................... 279
15.29.2.4 Displaying the Solution Graphically with MATLAB ........................................... 279
15.29.2.5 Displaying the Solution Graphically with OCTAVE ........................................... 281
15.29.2.6 Displaying the Solution Graphically with PYTHON ........................................... 281
15.29.3 Assessing Accuracy ........................................... 282
15.30 Exercises ........................................... 283
15.A Program(s) for FDM Approach to 1D Problem ........................................... 291
15.A.6 Listing of fdm1dla.c (C) ........................................... 291
15.B Evaluating Integrals for 1D FEM Problem ........................................ 292
  15.B.1 ... using MAXIMA .................................................... 292
  15.B.2 ... using MAPLE .................................................... 293
  15.B.3 ... using Mathematica ............................................. 293
15.C Program(s) for FEM Approach to 1D Problem .............................. 294
  15.C.6 Listing of fem1dla.c (C) ........................................... 294
15.D C Programs for FDM Approach to 2D Problems ............................. 295
  15.D.1 Listing of fdmwave1d.c ......................................... 295
  15.D.2 Listing of fdmdiffus1d.c ........................................ 297
  15.D.3 Listing of fdmlap2d.c ............................................ 299
  15.D.4 Listing of fdmlap2d_file.c ...................................... 300
15.E A Useful Integral .......................................................... 301
15.F Evaluating Integrals for 2D FEM Problem .................................. 302
  15.F.1 ... using MAXIMA .................................................... 302
  15.F.2 ... using MAPLE .................................................... 305
  15.F.3 ... using Mathematica ............................................. 308
15.G Program(s) for FEM Approach to 2D Problems ............................. 311
  15.G.6 Listing of fem2dla.c (C) ........................................... 311
Z Contacting Software Vendors .................................................... 315
Index ..................................................................................... 319
Chapter 1

Preliminaries

Over the past two decades, acquaintance with computational approaches to problems—and with the computational resources that facilitate those approaches—has come to be critically important for success in the sciences. This book aims to develop familiarity with a variety of computational tools and techniques in application particularly to problems in physics. Rather than selecting a single application program, we presume that productive use of contemporary computational resources requires acquaintance with several different sorts of tools, including

- an array processing program (e.g., IDL®, MATLAB®, OCTAVE, PYTHON, ...);
- a computer algebra system (e.g., MAPLE®, Mathematica®, MAXIMA, ...);
- a standard scientific programming language (e.g., FORTRAN, C, PYTHON, ...), both for programming ab initio and, more particularly, for creating driving programs to invoke commercially available subroutine packages like NUMERICAL RECIPES, and freely downloadable subroutine packages like LSODE, MUDPACK, and LAPACK; and
- a tool for graphical visualization of scalar and vector functions of one, two, and three independent variables (e.g., IDL, MATLAB, OCTAVE, PYTHON, MAXIMA, MAPLE, Mathematica, ...).

Further, to make effective use of these tools, the user must

- be acquainted with the main capabilities of at least one operating system (e.g., UNIX, Windows, Macintosh OS, ...),
- be fluent in the use of a text editor (e.g., gedit, xemacs, vim, winedt, ...) and of a program for creating drawings (e.g., tgif, ...), and
- of a publishing package (e.g., TexLive, IATEX, MiKTeX, OzTex, ...) capable of formatting elaborate equations, incorporating symbolic references within documents, generating tables of contents and indices, ... .

This book introduces intermediate-level physics students to a selected spectrum of these tools, helps them learn enough of the tools’ capabilities to know what the tools can do, and builds their confidence both in using the tools and in reading vendor-supplied documentation. Ultimately, we expect that

---

1 Many of the specific examples in this list are identified by names that are trademarks belonging to the vendor of the identified software and registered in the United States Patent and Trademark Office. Those that the author knows to have that status are identified with the symbol ® at the first occurrence of the name. Full contact information for the vendors of the software (and the owners of the trademarks) is compiled in Appendix Z.
students launched into the computational world as sophomores, say, will, as juniors and seniors, be motivated to use computational resources intelligently and successfully on their own initiative, whenever it seems to them appropriate to exploit those tools. As a resource, the computer should parallel the library; this book aims to help students develop the skills to support that view.\footnote{Uses of internet resources are conspicuously absent from the list of skills in this opening paragraph. While such uses are playing an increasingly important role both in education and in professional life, they are explicitly excluded from the purview of this book.}

In this book, the ultimate objective described in the previous paragraph is pursued in several steps:

1. You learn to manipulate the system you have and to work efficiently with whatever text editor is available. For the most part, this step is the task of the \textit{Local Guide}.

2. You learn the basic commands for one or more tools (how to start the tool, how to stop the tool, how to construct the primary entities—mathematical expressions, numerical arrays, \ldots—on which the tool works, how to manipulate those entities, how to generate output—both textual and graphical—from the tool, etc.). This step is the business of the first portion of this book and of the appendices.

3. You learn ways in which these tools can be used to advantage to address prototype problems in a variety of areas of physics. This step is the business of the second portion of this book, each chapter of which begins by describing several representative problems that involve a particular type of computation (solving ODEs, integrating, finding roots, \ldots). Then, each chapter addresses those problems with a succession of computational tools, some symbolic, some numeric—exploiting graphical displays whenever appropriate to the exercise at hand. Each chapter concludes with numerous exercises to direct your own further study of the tools and techniques addressed in the chapter.

You need not, of course, complete \textit{all} of one step before proceeding to \textit{some} of the next step. Once you have learned to manipulate your computer system and use an available text editor, you can pick and choose the tools and examples of greatest—or most immediate—interest to you. To be sure, some portions of earlier chapters are prerequisite to some portions of later chapters, but the linkages are neither deep nor extensive. Thus, you can hop around in this book as your needs and interests dictate.

In the remainder of this chapter, we address several general items relating to the design and use of computers and to the structure of this book. Here and there, specific items may well be site dependent. Thus, as a companion to this book, you must obtain from your local site administrator a copy of the \textit{Local Guide}, which supplements this book with detailed information that relates specifically to your site.

Be aware, in particular, that many of the chapters in this book are at least in part tutorial in nature. Full study of the material here presented requires you to replicate the illustrated “conversations” with the computer. To do so, you must—of course—be logged into an appropriate computer system, as described in the \textit{Local Guide}. This paragraph, however, is the \textit{only} point in the book at which the wisdom of being logged in is explicitly mentioned.

\subsection{An Orientation to Computers}

We begin by inventing (at least some aspects of) a computer, in the process motivating some of its main features and discussing briefly a few important underlying concepts and structures.
1.1.1 A Simple Responsive Machine

Consider first a typewriter. In broad outline, its user commands the printing mechanism (hereafter printer) to perform a desired sequence of actions by pressing the corresponding sequence of keys on the keyboard. Most keys cause the printer to print a particular character on the paper and advance the printhead to its next position. When the key labeled ‘a’ is pressed, for example, the character ‘a’ is printed on the paper and the printhead is advanced; when the shift key is held down while the key labeled ‘5’ is pressed (sometimes denoted (SHIFT/5)), the character ‘%’ is printed and the printhead is advanced; etc. A few keys command the printer to perform other actions. Pressing the space bar, for example, advances the printhead without printing a visible character. (Actually, it is useful to think that the space character, denoted ⟨SP⟩, has been “printed”.) Pressing the key labeled RETURN “prints” the carriage return character (denoted ⟨CR⟩), which moves the printhead to the beginning of the line and advances or feeds the paper one line further along.

We can, however, imagine a more general “typewriter”—i.e., a computer—in which an obedient and instructable “agent”—hereafter the central processing unit (CPU)—has been interposed between the keyboard and the printer. Further, let us build this expanded machine so that (a) pressing a key at the keyboard sends a (probably electrical) code identifying that key to the CPU and (b) the printer interprets and responds to each code received from the CPU. This machine reverts to our original typewriter if we tell the CPU to carry out or execute the statements or commands:

```
LOOP
  Read code from keyboard
  Send code to printer
END_LOOP
```

The action of the machine in response to representative key strokes would then be described as follows:

- When the key labeled ‘a’ is pressed, the keyboard sends the code for the character ‘a’ to the CPU, which then transmits that code to the printer.
- When the shift key is held down while the key labeled ‘5’ is pressed, the keyboard sends the code for the character ‘%’ to the CPU, which then transmits that code to the printer.
- When the space bar is pressed, the keyboard sends the code for the character ⟨SP⟩ to the CPU, which then transmits that code to the printer.
- When the key labeled RETURN is pressed, the keyboard sends the code for the character ⟨CR⟩ to the CPU, which then transmits that code to the printer.

In the first three cases, the printer displays the character identified by the received code and also advances the printhead. In the fourth case, the printer should both return the printhead and feed the paper. In fact, most printers treat returning the printhead and feeding the paper as two distinct operations. Receipt of the code for the character ⟨CR⟩ will effect the former operation; receipt of a different code, that for the line-feed character ⟨LF⟩, will effect the latter. While it is convenient to have a single keystroke at the keyboard accomplish both operations, most printers must receive two separate codes to accomplish the desired action. Thus, we must tell the CPU that receipt of the code for the character ⟨CR⟩ from the keyboard must trigger the sending of the codes for the pair of characters ⟨CR⟩⟨LF⟩ to the printer. To simulate a typewriter, we must embellish the above statements to:

---

3 The special words LOOP and END_LOOP bracket a group or block of instructions that are as a block to be executed repeatedly. We shall here ignore concerns about stopping the loop.

4 The special words IF, THEN, and END_IF convey a conditional execution of one or more statements. The statement(s) between the THEN and the END_IF will be executed only if the condition following the IF is true when the entire construction is encountered.
In this listing, we have introduced the word *program* to identify a complete set of instructions for the performance of some task, and we have introduced the special words *PROGRAM* and *END_PROGRAM* to bracket a program. We have also provided a way to designate an appropriate name for the program.

Note that, while a particular code is always associated with a character, not all codes are associated with *printing* characters. Non-printing characters are called *control* characters. When received by a printer (or other peripheral device), they result not in the display of a particular symbol but in the performance of some other function. We have already met ⟨CR⟩ and ⟨LF⟩. Other control characters familiar to the user of an ordinary typewriter are the backspace ⟨BS⟩, which causes the printhead to back up one space; the horizontal tab ⟨HT⟩, which causes the printhead to advance to the next pre-set (horizontal) tab position; and the vertical tab ⟨VT⟩, which advances the paper to the next preset (vertical) tab position.

### 1.1.2 Character Codes

To facilitate visualizing the codes seen by the CPU, imagine that the CPU receives its signals by “looking at” a row of eight light bulbs. Further, declare that pressing a particular key on the keyboard turns some of the bulbs on and leaves the rest off, and endow the CPU with a capacity to sense which bulbs are on and which off. If we represent a light bulb that is off by the symbol 0 and a light bulb that is on by the symbol 1, then we can convey a particular pattern by a string of eight 0’s and 1’s. The string 10011101, for example, represents the sequence on-off-on-on-off-on.

Although a particular pattern of 0’s and 1’s unambiguously conveys the character associated with it, it is useful to interpret this pattern alternatively as an integer in the *binary* (base-2) number system—a system in which only the two characters 0 and 1 are used to express numbers. In the more familiar decimal (*base-10*) number system, the digits of an integer reckoned from *right to left* are the 1’s digit (10 to the zero power), the 10’s digit (10 to the first power), the 100’s digit (10 squared), etc. Similarly, in the binary number system, the *bits* in an eight-bit integer, again reckoned from right to left, are the 1’s bit (2 to the zero power), the 2’s bit (2 to the first power), the 4’s bit (2 squared), the 8’s bit (2 cubed), the 16’s bit (2 to the fourth power), the 32’s bit (2 to the fifth power), the 64’s bit (2 to the sixth power), and the 128’s bit (2 to the seventh power). Just as the decimal integer 324 means
\[ 3 \times 10^2 + 2 \times 10^1 + 4 \times 10^0 \]
the binary integer 10011101 means
\[ 1 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \]
or, converting to decimal,
\[ 1 \times 128 + 0 \times 64 + 0 \times 32 + 1 \times 16 + 1 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1 = 157 \]

---

5The number eight is, of course, arbitrary but conventional. Because computers work internally in the binary (base-2) number system, powers of two—\(8 = 2^3\)—are especially convenient.

6Actually, the codes will be sent as a stream of *bits*, each of which is an electrical voltage level that will be either “high” or “low”, often said to be “on” or “off”.
The largest three-digit decimal integer is 999; the largest eight-bit binary integer is 11111111, which translates to the decimal integer 255.

An array of eight bits—called a byte—can assume 256 different patterns or values (00000000, 00000001, 00000010, ..., 11111111). Our choice of the byte for internal coding therefore permits us to distinguish 256 codes. Internally, the CPU sees only binary patterns (light bulbs that are on or off; electrical signals that are either high or low; areas on a magnetic tape that are either magnetized or unmagnetized; etc.), and these patterns are conveniently represented by sequences of bits. Externally, binary integers are cumbersome, so various more compact representations are often used. The binary pattern can be interpreted as a decimal integer (as above), but the conversion from binary to decimal is awkward. A more convenient but still compact notation involves grouping the bits in an eight-bit binary integer in the pattern xx-xxx-xxx and using the eight symbols 0, 1, 2, ..., 7 to represent the three-bit binary integers 000, 001, 010, 011, 100, 101, 110, and 111. The integer 10011101, for example, would then have the translation

\[10011101 = 10-011-101 = 235\]

into this octal (base-8) number system. (The first grouping has only two bits and hence can have only the values 0, 1, 2, or 3.) Here, the octal integer 235 is interpreted in decimal as \(2 \times 8^2 + 3 \times 8^1 + 5 \times 8^0 = 2 \times 64 + 3 \times 8 + 5 \times 1 = 157\). The largest eight-bit binary integer 11111111 has the representation 377 in octal. This is, of course, the same integer as 255 (decimal).

A still more compact representation of an eight-bit binary integer involves dividing the byte into two four-bit nybbles. Then, with the representation 0000=0, 0001=1, 0010=2, 0011=3, 0100=4, 0101=5, 0110=6, 0111=7, 1000=8, 1001=9, 1010=A, 1011=B, 1100=C, 1101=D, 1110=E, and 1111=F, the binary integer can be represented by two “digits”. For example, the integer 10011101 = 1001-1101 = 9D. This representation expresses the integer in a base-16 or hexadecimal number system. The largest eight-bit binary integer 11111111 has the translation FF into hexadecimal, a value to be compared with 377 in octal and 255 in decimal.

### 1.1.3 The ASCII Character Set

The code transmitted by a particular key on the keyboard is determined by the electrical structure of the keyboard, not by the label on the key. A given key transmits a particular code regardless of the label on the key. Likewise, a code received by a printer identifies, for example, a particular orientation of the printwheel regardless of what character happens to be embossed on the finger at that position. The codes merely identify positions on the keyboard or orientations of the printwheel; no code has any necessary connection with any particular character, and in some contexts associations other than the conventional are adopted.

There are, however, a number of conventional associations of codes with characters. The most commonly used scheme is the American Standard Code for Information Interchange (ASCII, pronounced ass′-key). In this code, characters are associated with eight-bit binary patterns. While the second 128 of the 256 distinguishable patterns [i.e., characters 128–255 (decimal)] have a variety of assignments to characters, the first 128 patterns [i.e., characters 0–127 (decimal)] have the standard assignments enumerated in Table 1.1. The control characters (non-printing characters) all have (decimal) ASCII codes in the range 0–31. Further, the ASCII code for each uppercase letter is 32 less than the code for the corresponding lowercase letter; i.e., turning off the 32-bit in the code for a lowercase letter generates the code for the corresponding uppercase letter. Finally, the ASCII code for a control character is 64 less than the code for the associated uppercase letter; i.e., turning off the 64-bit in the code for an uppercase letter (say C) generates the code for the corresponding control character (CTRL/C). Numerical digits occur in ascending order and before the characters in the alphabet; punctuation marks and other symbols (+, -, *, /, @, [, ..., ] are distributed where the previous assignments leave gaps.
1.1.4 Representation of Data in a Computer

A computer consisting of no more than a keyboard, a CPU with only the above described capabilities, and a printer would, of course, be of little value. Let us expand our computer by adding an internal storage capacity (memory and auxiliary hard disks\(^7\)) consisting of individual cells, each identified by its address, which simply counts the cell’s position from the first cell, and each capable of storing (the code for) a single character. Further, let us endow the CPU with an ability to write codes to and read codes from individual cells in this memory. We understand that a (new) code written into a cell always replaces or overwrites the (previous) contents of that cell, thereby rendering the previous contents no longer retrievable. We declare, however, that reading a code from a cell does not change the contents of the cell.

Typical present-day computers will have a capacity to store an enormous number of bytes—gigabytes, even terrabytes—of information. As we have described it so far, each byte stores an eight-bit pattern of 0’s and 1’s, each pattern being associated with a particular (printing or control) character. The association with characters, however, is not the only possible interpretation of the information stored in one or more bytes of a computer’s memory. Several other interpretations are necessary. Beyond the association of eight-bit patterns with characters (and successions of such patterns with character strings), the CPU might represent integers of various sizes by interpreting

- an eight-bit byte as an unsigned eight-bit integer, assigning its 256 different patterns to the (positive) integers ranging (in decimal) from 0 to 255.

- an eight-bit byte as a signed eight-bit integer, assigning its 256 different patterns to the (negative and positive) integers ranging (in decimal) from −128 to +127. (The range is not symmetric because we must assign one of the patterns to the integer 0.) The highest order bit normally conveys the sign of the value and the remaining seven bits convey the value, though the connection between bit patterns and values—especially negative values—is not always as straightforward as one might naively assume.\(^8\)

- a sixteen-bit combination of two consecutive bytes as an unsigned sixteen-bit integer, with its \(256^2 = 65536\) values assigned to the (positive) integers ranging (in decimal) from 0 to 65536.

- a sixteen-bit combination of two consecutive bytes as a signed sixteen-bit integer, with its 65536 values assigned to the (positive) integers ranging (in decimal) from −32768 to +32767.

- a 32-bit combination of four consecutive bytes as an unsigned 32-bit integer, with its \(256^2 = 4294967296\) values assigned to the (positive) integers ranging (in decimal) from 0 to 4294967296.

- a 32-bit combination of four consecutive bytes as a signed 32-bit integer, with its 4294967296 values assigned to the (positive) integers ranging (in decimal) from −2147483684 to +2147483683.

Some architectures even use 64-bit unsigned and signed integers to expand the range of available integers even further.

Especially for scientific computations, integers alone will not suffice. Computers provide for storage of numbers with decimal points and exponents by designing the CPU to interpret

- a 32-bit combination of four consecutive bytes as a single-precision floating point number. In the IEEE standard for this format, eight bits (one byte) are assigned to store the proper

\(^7\)For purposes of this discussion we will ignore the very considerable differences between (volatile) memory and (non-volatile) hard disks.

\(^8\)Negative values are frequently stored in what is called two’s-complement form, a discussion of which is beyond the needs or scope of this book. (The two’s complement notation is adopted because it simplifies algorithms that perform arithmetic on signed integers.)
Table 1.1: The ASCII character codes. In this table, the first column in each pair lists the decimal code for the character that is identified in the second column of each pair.

<table>
<thead>
<tr>
<th>Code</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NUL</td>
</tr>
<tr>
<td>16</td>
<td>DLE</td>
</tr>
<tr>
<td>32</td>
<td>SP</td>
</tr>
<tr>
<td>48</td>
<td>0</td>
</tr>
<tr>
<td>64</td>
<td>@</td>
</tr>
<tr>
<td>80</td>
<td>P</td>
</tr>
<tr>
<td>96</td>
<td>p</td>
</tr>
<tr>
<td>112</td>
<td>p</td>
</tr>
<tr>
<td>16</td>
<td>SOH</td>
</tr>
<tr>
<td>17</td>
<td>DC1</td>
</tr>
<tr>
<td>33</td>
<td>!</td>
</tr>
<tr>
<td>50</td>
<td>2</td>
</tr>
<tr>
<td>52</td>
<td>4</td>
</tr>
<tr>
<td>53</td>
<td>5</td>
</tr>
<tr>
<td>68</td>
<td>D</td>
</tr>
<tr>
<td>84</td>
<td>T</td>
</tr>
<tr>
<td>100</td>
<td>d</td>
</tr>
<tr>
<td>116</td>
<td>t</td>
</tr>
<tr>
<td>23</td>
<td>ETB</td>
</tr>
<tr>
<td>39</td>
<td>'</td>
</tr>
<tr>
<td>55</td>
<td>7</td>
</tr>
<tr>
<td>71</td>
<td>G</td>
</tr>
<tr>
<td>87</td>
<td>W</td>
</tr>
<tr>
<td>103</td>
<td>g</td>
</tr>
<tr>
<td>119</td>
<td>w</td>
</tr>
<tr>
<td>30</td>
<td>RS</td>
</tr>
<tr>
<td>45</td>
<td>-</td>
</tr>
<tr>
<td>61</td>
<td>=</td>
</tr>
<tr>
<td>77</td>
<td>M</td>
</tr>
<tr>
<td>93</td>
<td>]</td>
</tr>
<tr>
<td>109</td>
<td>m</td>
</tr>
<tr>
<td>125</td>
<td>]</td>
</tr>
<tr>
<td>30</td>
<td>RS</td>
</tr>
<tr>
<td>46</td>
<td>,</td>
</tr>
<tr>
<td>62</td>
<td>&gt;</td>
</tr>
<tr>
<td>78</td>
<td>N</td>
</tr>
<tr>
<td>94</td>
<td>,</td>
</tr>
<tr>
<td>110</td>
<td>n</td>
</tr>
<tr>
<td>126</td>
<td>~</td>
</tr>
</tbody>
</table>

Power or 2 (as an eight-bit signed integer), one bit is assigned to store the sign (0=+, 1=−) of the value, and 23 bits (three bytes minus the sign bit) are assigned to store the digits of the (absolute value of the) value itself. In this format, values ranging (in decimal) from \(1.175 \times 10^{-38}\) to \(3.403 \times 10^{+38}\) can be represented, though only to a precision of about six decimal digits.\(^9\)

- a 64-bit combination of eight consecutive bytes as a double-precision floating point number.

In the IEEE standard for this format, eleven bits are assigned to store the exponent (as an eleven-bit signed integer, one bit is assigned to store the sign (0=+, 1=−) of the value, and 52 bits are assigned to store the digits of the (absolute value of the) value itself. In this format, values ranging (in decimal) from \(2.225 \times 10^{-308}\) to \(1.798 \times 10^{+308}\) can be represented, though only to a precision of about fifteen decimal digits.

Must computer architectures conform to these standards. Further, many computers make available one or more extended floating-point formats of their own design.

Clearly, many different data types are in common use. Most importantly, the information stored in a particular byte or aggregate of bytes contains nothing at all to identify its data type. The four bytes of a character string are indistinguishable from the four bytes in a 32-bit unsigned integer and both are indistinguishable from the four bytes in a single-precision floating point number. The bit pattern in those four bytes can be interpreted in any of these ways (and in others as well). It is the programmer’s responsibility to make sure that the program treats stored values in a way appropriate to their data types, usually by referring to memory cells with names that convey the data type. When conversion from one form to another—e.g., character to associated numerical ASCII code—is necessary, the programmer must invoke an appropriate routine to effect the conversion.

\(^9\)Note that the use of an explicit bit to convey the sign of the value means that there are in this format two zeroes. Plus zero is different from minus zero!
1.2 Files and Directories

At (nearly) the most microscopic level, information in a computer is recorded in bytes stored in memory or, more permanently, on a hard drive. At the next level up, aggregations of these bytes into larger units that must be kept together are called files. Each file will have a name. Some of the files containing portions of or referenced in this book, for example, are named `assemble.tex`, `laplace.f`, `laplace.c`, `trapezoidal.xc`, and `diffract.ps`. The part of the name before the dot conveys something of what the file contains; when used, the part after the dot—the extension or file type—conveys the type of file.\(^{10}\) Some files—called ASCII text files—contain nothing but printing ASCII characters (and perhaps such simple control characters as \(<\text{CR}>\) and \(<\text{HT}>\)) and can be displayed on the screen, printed on a printer, or examined and edited with a text editor. Though some of their bytes can be interpreted as printing characters, other files—called binary files—contain also (perhaps numerous) non-printing characters and cannot be displayed on the screen, examined in (ordinary) text editors, or printed on a printer. Files of this latter type may be special data files created by programs; more often, they are executable files which contain compiled programs, and the bit patterns stored in the file are intended to be interpreted as instructions to the CPU. Whatever the type of file and the nature of the bit patterns it contains, each file is a unit whose component bytes must be kept together as a single entity.

Any computer system will, of course, store a very large number of individual files. To keep these files under some semblance of control, they will commonly be grouped together into aggregates of various sizes, those aggregates will themselves be assembled into higher-level aggregates, **those** into still higher-level aggregates, \ldots. The process is analogous to the aggregating of individual documents into a file folder, of these folders into file drawers, of the drawers into file cabinets, \ldots. In the computer world, we need then not only the files themselves but a new type of file that basically lists the contents of the aggregate that it represents. The resulting structure for keeping track of files looks like a tree. At the highest level, the tree has a single file—the root directory—that contains the names of the files it contains (and information about their locations on the disks of the computer). Some of those files may themselves describe (sub)directories, in each of which are listed the names of the files it contains. Some of those files in turn may describe (subsub)directories.\(^{11}\) Locating a specific file in the entire structure then requires not only giving the file name but also describing its path—the sequence of directories through which we must pass from the root directory to reach the file. In UNIX, the root directory for the entire storage system is named `\(/\)`; the forward slash is also used to separate directory paths in an extended path. Thus, to specify the location of a file buried several directories down from this universal starting point, we would have to supply an identifier like

```
/usr/people/cook/CCLI/intro/intro.tex
```

which indicates that the file `intro.tex`—the \LaTeX source file for this chapter—will be found in the `intro` directory in the `CCLI` directory in the `cook` directory in the `people` directory in the `usr` directory in the `\(/\)` (root) directory of the computer system in which it resides.

In the previous paragraph (and in the rest of this book), we use UNIX style file specifications. The corresponding specifications appropriate to the computing system(s) available at your site are described in the Local Guide.\(^{12}\) That document also explains conventions about (and restrictions imposed on) file names and types, user accounts, and other matters that vary so much from site to site that this book cannot sensibly explain them all.

---

\(^{10}\)The extension `.tex` conventionally identifies a \TeX source file; `.f` and `.c` identify files containing source code for FORTRAN and C programs; in this book, `.xf` and `.xc` identify executable files generated when FORTRAN and C programs are compiled; and `.ps`, `.eps`, and `.pdf` identify PostScript and PDF files.

\(^{11}\)Since directories may ultimately be buried many levels deep, we shall suppress the multitude of sub's that might appear, understanding that the simpler word ‘directory’ will refer to a directory without regard to its position in the overall hierarchy.

\(^{12}\)In Windows, for example, the backslash character `\` is used to separate directories in a path.
1.3 Operating Systems

Underneath it all, everything that a computer does is controlled by its operating system, which makes available a variety of standard commands to instruct the computer to carry out common tasks. In some cases, the user invokes a command by typing its name (and any necessary arguments) in a text-entry window or command-line interface (CLI). In other cases, the user clicks a mouse button on an icon or drags an icon to a new location on the desktop in a graphical user interface (GUI). However a command is conveyed to the operating system, it at base simply invokes a program that carries out the selected task and then returns control to the operating system for the next command. At the very minimum, the operating system must make available commands for

- logging in and logging out, paying attention on multi-user systems to user authorization (normally controlled through usernames and passwords).
- setting and changing the default directory, which is the directory to which file names refer when no path is specified.
- copying a file to another directory or deleting it altogether from its current directory.
- establishing various levels of file protection file by file and changing those specifications.
- creating ASCII files through the use of a text editor.
- customizing the user’s environment through the creation of environment variables, aliases, and other shorthands.
- retrieving and editing a previously executed command before it is submitted again for execution.
- displaying a file on the screen.
- printing a file to a printer.
- copying a selected portion of the screen to a file.
- converting files from one format to another.

The details of the ways in which these several capabilities are invoked and conventions about assigning user names, passwords, and default directories vary considerably among operating systems and are, even with the same operating system, site-specific. The Local Guide for your site describes those details.

1.4 Glossary, Conventions, and Understandings

In this section, we enumerate and define a number of terms to be used throughout this book, and we make a variety of observations that otherwise would have to be repeated several times.

- Typographically, we use the typewriter font for all program listings and for command lines displayed in the text. We also use this font for command and function names embedded in the text itself without enclosing these names in quotation marks (unless the absence of quotation marks creates ambiguity or confusion).
- In describing mouse operations, we use ML, MM, and MR for the left, middle, and right mouse buttons, respectively.\(^{13}\) The Local Guide explains how to translate these symbols if your mouse has fewer than three buttons.

\(^{13}\)If you have invoked a feature of your operating system that permits reversing the conventional association of mouse buttons with actions, then you will have to read our MR to mean your ML, etc.
As a shorthand, we use the phrase ‘Select …’ for the operations of moving the cursor over the indicated item (which may involve pulling down a menu) and then clicking ML.

We use italic type for window names, SMALL CAPS to identify menus, and single quotation marks to enclose the names of buttons or menu items. Thus, for example, in a tutorial segment, we might instruct you to “Select ‘Print’ from the FILE menu in the WinEdit window”.

The lines dividing statements from commands from instructions are difficult to draw. In this book, we strive to refer to a complete instruction in some programming language as a statement and to reserve the word command for the keyword that introduces a statement. For example, we would speak of the command integrate but refer to the construction

\[
\text{integrate}(\sin(k\cdot x), x, 0, \frac{\pi}{2})
\]

as a statement. Even this distinction is difficult to draw, however, because statements can be nested to produce compound statements that could, with justification, themselves be referred to as statements.

The lines dividing functions from procedures from subroutines are also difficult to draw. Indeed, some computer languages regard these terms as synonymous. When a distinction is made, a function is a construction which, when executed, accepts arguments as input but returns a value to the variable(s) to which the function is assigned; the function \text{SQRT}, for example, would be invoked with a statement like

\[
R = \text{SQRT}(x^2 + y^2)
\]

Procedures and subroutines, however, (normally) have only arguments, some of which will supply input and others of which name the variables into which returned values will be placed; a procedure—call it \text{SQRTPRO}—to return in its second argument the square root of its first argument would be invoked with a statement like

\[
\text{SQRTPRO}(x^2 + y^2, R)
\]

having no variable or equal sign at its beginning.\footnote{In some languages, procedures can optionally be written \text{err = SQRTPRO}(x^2 + y^2, R), in which case the procedure returns a value to the variable \text{err} to convey that the procedure encountered a problem in its execution. Testing \text{err} after the procedure is invoked can then be used to trap errors and alert you to possible incorrect output.}

The conventional file type for C programs is .c or .cc. We use .c in this book.

Most of the statements presented in this book can be submitted as they stand and executed by the program in whose command language the statement is written. Occasionally, we illustrate the general format of a statement without being sufficiently explicit to render the statement executable. Statements in the former category will be preceded by the appropriate prompt; statements in the latter category will be presented without a prompt. As a general rule, statements preceded with a prompt can—and should—be executed as you work your way through the material. Statements without a prompt should not, and most often could not, be executed.

Especially in constructing statements for computer algebra systems and presenting their output, we will not always present the output in exactly the form or with exactly the appearance it will actually have. In particular, we will frequently use unsubscripted variables, e.g. \text{x} or \text{xf}, in the statement to be executed but render these variables as subscripted, e.g., \text{x}_1 or \text{x}_f, in the displayed output.
• At various points in this book, we define, use, and/or refer to a variety of files specific to the text. All such files are stored on your local computer system and are available for your use. The head of the directory tree in which those files are stored is identified explicitly in the Local Guide but is referred to throughout this book with the symbol $\text{HEAD}$. In any attempt to access these files, you must replace this symbol in the text with whatever is proper for your site. More specifically, files associated with the chapter on IDL will be found in the directory $\text{HEAD/idl}$, those associated with the chapter on integration will be found in subdirectories of the directory $\text{HEAD/integ}$, etc.

• Less often, we will also refer to items in the directory tree associated with one or another application program. The heads of those directories are also site specific and are identified in the Local Guide. In this book, we refer to the head of the IDL directory tree with the symbol $\text{IDLHEAD}$, the head of the Numerical Recipes directory tree with the symbol $\text{NRHEAD}$, etc.

• Especially in the chapters on solving ordinary differential equations, evaluating integrals, and finding roots (but also occasionally in other chapters), the sequence of numbers assigned to the end of chapter exercises has gaps. Most of these gaps come about because those few exercises that are specific to a particular software package are omitted in versions that do not include that package. The gaps are placed so that the numbers assigned to identical exercises in various versions of this book will be the same, thereby facilitating the keying of solutions to representative exercises to the book itself.

• In a similar vein, to facilitate communication among users of different versions, chapters and sections focusing on a particular software package retain in all versions the same numbers. That decision, of course, means that the sequence of numbers assigned to chapters and sections in a particular version will have gaps corresponding to chapters and sections omitted. In contrast, equation numbers, figure numbers, and table numbers will run continuously within chapters and page numbers will run continuously from the beginning of the book to the end. Thus, references to chapter numbers and section numbers will be universal, the same in all versions, while references to equations, figures, tables, and pages will be version-specific.

• With regard to the index, please note the following:

  - Italic page numbers in the index convey that the page reference is to an end-of-chapter exercise, not to a textual discussion.
  - Because index commands in \LaTeX{} cannot be placed in \texttt{verbatim} environments and, for a few similar reasons, some items referred to in the index may actually occur towards the bottom of the previous page or near the top of the following page. Readers must therefore understand that an index entry pointing to page 215, for example, identifies an area that includes perhaps the bottom quarter of page 214 and the top quarter of page 216 as well. Straightening out this departure from the conventional understanding is difficult; it is much easier to adjust the understanding of the meaning of a page reference.
  - References to such symbols as \texttt{.*}, :, ^, ==, /\texttt{*...*/}, and ! are included in the index. Since many of these symbols are, however, extremely common, the index includes pointers to only some of their appearances within the text. Those included have been chosen to illustrate a representative spectrum of uses, so reviewing the few items identified in the index should convey the full range of uses of these symbols.

1.5 Assumed Background

Mathematically, we assume in this book that you understand the notions of derivatives, integrals, and ordinary differential equations and that you have some acquaintance with linear algebra (matrix operations, eigenvalues and eigenvectors, \ldots). Physically, we suppose that you have taken a couple of introductory, calculus-based courses in physics and are continuing with intermediate courses in
physics. This book makes contact with many intermediate-level physics courses, but it is not focused on any particular one of those courses. It draws on topics covered in several such courses whenever appropriate.

In addition, we include here a brief discussion of two mathematical topics that are necessary for some of what follows but that may well not have been treated in any of the courses viewed as prerequisite for the study of this book.

1.5.1 The Gamma Function

The factorial function 
\[ n! = n \times (n-1) \times (n-2) \times (n-3) \times \cdots \times 2 \times 1 \]  
(1.1)
is usually familiar. The double factorial function 
\[ n!! = n \times (n-2) \times (n-4) \times (n-6) \times \cdots \times 1 \]  
(1.2)
is less familiar but occurs frequently as well. Whether \( n \) is even or odd, the double factorial can be recast in terms of single factorials. If, for example, \( n \) is even, say 10, we can recast its double factorial in the form
\[
10!! = 10 \times 8 \times 6 \times 4 \times 2 = 2^5 \times 5 \times 4 \times 3 \times 2 \times 1 = 2^5 5!
\]
\[
\Rightarrow (2n)!! = 2^n n!
\]
(1.3)
If \( n \) is odd, say 9, recasting its double factorial takes mildly more work, but is illustrated in the chain
\[
9!! = 9 \times 7 \times 5 \times 3 \times 1 = \frac{9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1}{8 \times 6 \times 4 \times 2} = \frac{9!}{8!!} = \frac{9!}{2^4 4!}
\]
\[
\Rightarrow (2n + 1)!! = \frac{(2n + 1)!}{2^n n!}
\]
(1.4)

We might wonder whether it is possible to define a function of a continuous variable that will coincide with the factorial function when its argument is an integer. Consider the function \( \Gamma(\nu) \) defined by the integral
\[
\Gamma(\nu) = \int_0^\infty t^{\nu-1} e^{-t} \, dt
\]
(1.5)
We quickly conclude that
\[
\Gamma(1) = \int_0^\infty e^{-t} \, dt = 1
\]
(1.6)
With a little more effort, we can evaluate \( \Gamma(1/2) \). We begin by writing the definition of \( \Gamma(1/2) \) and introducing the new variable \( x^2 = t \) in the definition, finding that
\[
\Gamma\left(\frac{1}{2}\right) = \int_0^\infty \frac{e^{-t}}{\sqrt{t}} \, dt = \int_0^\infty e^{-x^2} \frac{x}{\sqrt{x}} \, dx = \int_0^\infty e^{-x^2} \, dx
\]
(1.7)
Then, we examine the square of the quantity of interest, change to polar coordinates, and find that
\[
\left[ \Gamma\left(\frac{1}{2}\right) \right]^2 = \left( \int_0^\infty e^{-x^2} \, dx \right)^2 = \left( \int_0^\infty e^{-x^2} \, dx \right) \left( \int_0^\infty e^{-y^2} \, dy \right)
\]
(1.8)
\[
= \int_0^\infty \int_0^\infty e^{-(x^2+y^2)} \, dx \, dy = \int_0^\infty e^{-r^2} \, r \, dr \int_0^{2\pi} \, d\phi = \pi
\]
(1.9)
and we conclude that\[\Gamma \left( \frac{1}{2} \right) = \sqrt{\pi} \] (1.10)

Evaluation of the Gamma function at most other arguments must be done numerically.

The Gamma function, however, has a particularly interesting property that we can deduce if we apply integration by parts to the definition. Provided \( \nu > 1 \), we find that
\[
\Gamma(\nu) = -\int_0^\infty t^{\nu-1} e^{-t} \, dt = (\nu - 1) \int_0^\infty t^{\nu-2} e^{-t} \, dt = (\nu - 1) \Gamma(\nu - 1) \quad (1.11)
\]

Applying this recursion relationship when the argument of the Gamma function is an integer, we find, for example, that
\[
\Gamma(5) = 4 \Gamma(4) = 4 \times 3 \Gamma(3) = 4 \times 3 \times 2 \Gamma(2) = 4 \times 3 \times 2 \times 1 \Gamma(1) = 4 \times 3 \times 2 \times 1 = 4! \quad (1.12)
\]

More generally, a similar argument leads to the conclusion that
\[
\Gamma(n + 1) = n! \quad (1.13)
\]

and we have indeed succeeded in finding a function that is the natural extension of the factorial function to non-integral arguments. Indeed, one often sees the notation \( \nu! \) as an alternative to the notation \( \Gamma(\nu + 1) \)—and the latter in fact provides a formal definition of the former.\footnote{The requirement at Eq. (1.11) that \( \nu > 1 \) limits the range of \( \nu \) for which the integral is acceptable as a definition of the Gamma function. Outside that range, we simply take the recursion relationship itself to define the function, so the recursion relationship is always valid while the integral converges only for \( \nu > 1 \).}

Note that, since we know quite explicitly that \( \Gamma(1) = 1 \), this connection between the Gamma and factorial functions supports what is sometimes an assertion of convenience, namely that \( 0! = 1 \).

### 1.5.2 The Laplace Transform

One tool used behind the scenes by symbolic solvers of ordinary differential equations is called the Laplace transform, which we describe here to avoid duplicating the discussion at several places in subsequent chapters. While we are not likely to make much use of the Laplace transform directly, knowing its properties may sometimes be valuable as we try to guide a symbolic manipulator that uses the technique. Defined for a function \( f(t) \) by the integral
\[
\mathcal{L}\left(f(t)\right) = \tilde{f}(s) = \int_0^\infty e^{-st} f(t) \, dt \quad (1.14)
\]

this transform has several important properties:

- The Laplace transform of a linear combination of functions is that same linear combination of the Laplace transforms of the separate functions,
\[
\mathcal{L}\left(af(t) + bg(t)\right) = \int_0^\infty e^{-st} \left(af(t) + bg(t)\right) \, dt
\]
\[
= a \int_0^\infty e^{-st} f(t) \, dt + b \int_0^\infty e^{-st} g(t) \, dt
\]
\[
= a \mathcal{L}\left(f(t)\right) + b \mathcal{L}\left(g(t)\right) \quad (1.15)
\]
i.e., in more technical terminology, \( \mathcal{L} \) is a linear operator (because integration itself is a linear operation).
Table 1.2: A short table of Laplace transforms.

<table>
<thead>
<tr>
<th>$f(t)$</th>
<th>$\tilde{f}(s)$</th>
<th>$f(t)$</th>
<th>$\tilde{f}(s)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t^n$</td>
<td>$\frac{n!}{s^{n+1}}$</td>
<td>$\sin \omega t$</td>
<td>$\frac{\omega}{s^2 + \omega^2}$</td>
</tr>
<tr>
<td>$\cos \omega t$</td>
<td>$\frac{s}{s^2 + \omega^2}$</td>
<td>$e^{at}$</td>
<td>$\frac{1}{s-a}$</td>
</tr>
<tr>
<td>$\frac{dx}{dt}(t)$</td>
<td>$s \tilde{x}(s) - x(0)$</td>
<td>$\frac{d^2x}{dt^2}(t)$</td>
<td>$s^2 \tilde{x}(s) - s x(0) - \frac{dx}{dt}(0)$</td>
</tr>
</tbody>
</table>

- The Laplace transform of the first derivative of a function $f(t)$ is simply related to the Laplace transform of $f(t)$. We need merely integrate the formal expression for the transform of the derivative by parts to find that
  \[
  \frac{df}{dt}(s) = \int_0^\infty e^{-st} \frac{df}{dt}(t) \, dt = e^{-st} f(t) \bigg|_0^\infty + s \int_0^\infty e^{-st} f(t) \, dt = s\tilde{f}(s) - f(0) \quad (1.16)
  \]

- The Laplace transform of a higher-order derivative is also simply related to the Laplace transform of the original function. We merely apply the identity in Eq. (1.16) repeatedly. The Laplace transform of a second derivative, for example, has the evaluation
  \[
  \frac{d^2f}{dt^2}(s) = s \frac{df}{dt}(s) - \frac{df}{dt}(0) = s \bigg( s\tilde{f}(s) - f(0) \bigg) - \frac{df}{dt}(0) = s^2 \tilde{f}(s) - sf(0) - \frac{df}{dt}(0) \quad (1.17)
  \]

As we shall see particularly in the chapter on ordinary differential equations, these last two properties, which convert differential expressions involving $f(t)$ into algebraic expressions involving $\tilde{f}(s)$, can be extended to convert some types of differential equations into algebraic equations. As a consequence, we anticipate that the Laplace transform may well play an important role in some approaches to solving ordinary differential equations.

Provided we can actually do the integral in Eq. (1.14), we can, of course, supplement these general properties by explicit evaluation of any number of Laplace transforms. Each entry in Table 1.2—a very short table of Laplace transforms—was obtained by explicit evaluation of the defining integral for the corresponding function.

1.6 Licensing Issues

Much of the software on every device in computational facilities around the world is proprietary and subject to the provisions both of the applicable copyright laws and of license agreements between the local institution and the vendors of the software. Usually—but not always, the licenses acquired by a given institution will permit simultaneous use on all of the devices in a laboratory at that institution. Almost certainly, the licenses limit use to projects and activities at that institution and prohibit copying of the software, except for purposes of system maintenance and backup. All users of all devices must be constantly mindful of the proprietary nature of much of the available software and must abide by the restrictions imposed by the copyright laws and by the license agreements. Those restrictions for each software package available at your site are described in the Local Guide.
Chapter 9

Introduction to Programming

In this chapter, we address the task of composing sets of statements\(^1\) that will cause an obedient, efficient, and very literal servant (i.e., computer) to perform exactly the task intended by the master (i.e., programmer). A general strategy by which a particular task can be accomplished in a finite (though perhaps large) number of steps is called an algorithm. A specific expression of an algorithm in a suitable language is called a program. Hence, this chapter is about designing algorithms and implementing them in programs. More specifically, it is about designing algorithms for the performance of tasks that will ultimately be assigned to a computer and about implementing those algorithms in several possible computer languages.

Designing algorithms and implementing them in programs that direct a computer to perform various tasks is really not very different from designing algorithms and implementing them in programs that direct a baker to bake a cake, a knitter to knit a sweater, or a cab driver to drive from the airport to the hotel. In the aggregate, an algorithm will obtain all necessary inputs, manipulate those inputs in some way, and produce the required outputs. Each step in the process must be systematically specified using elementary statements, each of which means only one thing to the servant (person or computer) that will perform the task. Some of the appropriate elementary statements—particularly the action statements that specify simple actions—will vary with the general type of task to be performed. “Mix” and “whip”, for example, are among the elementary action statements that must be understood by the baker of cakes; “knit”, “purl”, and “cast on” must be part of the vocabulary of a knitter of sweaters; while “turn right” and “follow interstate 90 west” are action statements for the cab driver.

The performance of even simple tasks, however, entails the execution of suitable action statements in the right order. The complete description of an algorithm must therefore indicate not only which actions are to be performed but also the order of their performance. Those aspects of an algorithm that specify the order of performance of action statements are called control structures.

A language for specifying algorithms must therefore provide not only elementary action statements, which vary with the general type of task to be performed, but also control structures, which are more universal than the action statements. In the first two sections of this chapter, we focus particularly on identifying fundamental control structures and illustrating their role in several simple algorithms. In later sections, we explain how the general structures introduced in the first section are implemented in standard languages. More detailed information about various languages may be found in the references listed in Section 9.17.

\(^1\)Individual instructions in a computer language are usually called statements, and we use that word throughout this book.
9.1 Components of a Programming Language

While we could code programs directly in the binary language that computers use internally, the process would be slow, tedious, prone to error, and resistant to debugging. As an alternative, a wide assortment of high-level languages has been developed. Using a chosen high-level language, the programmer creates an ASCII file containing the source code, which lays out what the computer must do to accomplish the desired task. Then, that source code must be translated into machine code—a task that a computer equipped with the proper translating program can do for itself.

Actually, there are two approaches to this translation. In the first, the computer interprets the source code, which means that each statement in the code is translated by the computer each time the statement is executed; execution is slow but the process for us is simple: we write the source code and run the program. In the second approach, the computer first compiles the source code into binary object code, and then links the object code with an assortment of system routines and possibly additional user-written routines. Only after compilation and linking have been completed can the resulting binary or executable file actually be run. This second approach is more complicated for us, since we must both write the source code and then compile and link it. The advantages are twofold: (1) the final executable file is stored in the machine and can be run any number of times without repeating compilation and linking; and (2) execution of compiled code is faster than execution when the code is interpreted.

The prospective user of a computer need only learn how to construct the source code in whatever language is to be used. While the detailed syntax of the statements that can be constructed depends very much on the language chosen, certain common elements can be identified. In particular, all languages provide ways to make use of computer memory, to obtain input and display output, and to control the flow of execution within the program. In this section, we describe those common elements as a prelude to laying out the details for specific languages in later sections.

9.1.1 Variables, Variable Names, and Use of Memory

Computers provide a means for working with data. The data, however, must be accessible to the central processing unit (CPU), which is the part of the hardware that directs and controls all actions carried out by the computer. Let us, therefore, endow the CPU with a capacity to reserve an available memory cell (or a contiguous succession of them if one cell is not large enough) and assign to it (them) a variable name, which we specify. Throughout the execution of the program, the assigned name then refers to this unique cubbyhole in the machine’s memory.

While the details vary from language to language, we suppose here that we can choose variable names freely, subject only to the constraints that

- the first character must be a letter (a–z; A–Z),
- subsequent characters can be chosen from the letters (a–z; A–Z), the digits (0–9), and the underscore character (_), and
- the several special words (keywords, e.g., IF, DO, UNTIL, ...) that we shall introduce in subsequent sections are reserved (or protected) and cannot be used as variable names.

While we do not explicitly impose a length limit on variable names, shorter names are preferred, simply because longer names take longer to type, take more space in the line, and, taken to extremes, result in program listings that are difficult to read. Generally, of course, we want to choose variable names that help us remember the significance of the quantity the name identifies.

\footnote{Some languages are case-sensitive and others are not. For our general discussion, we shall assume case insensitivity, so a and A are indistinguishable in variable names.}
9.1. COMPONENTS OF A PROGRAMMING LANGUAGE

Even in the generic discussion of this section, however, we shall pay attention to the data type of each quantity represented. In Chapter 1, we distinguished floating point numbers, integers, and character strings from one another in several ways. In structuring programs, we must remain always aware of this distinction. Further, the computer in interpreting or compiling our source code must either make assumptions or be told the data type of each variable we use. Some languages (e.g., Pascal, C) require an explicit declaration of variable names and associated data types before the name can be used. Other languages (e.g., BASIC, FORTRAN) exploit implicit data typing by looking to the composition of the variable name to determine the intended data type—though more recent versions of FORTRAN, for example, also admit explicit data typing. Still others (e.g., IDL, MATLAB, OCTAVE) assign a data type dynamically and automatically on the basis of the value assigned to the variable. In our general discussion, we shall adopt a convention that keeps us consciously aware of data type every time we use a variable name. Following the pattern actually used in BASIC, we shall take variable names with no appended suffix to represent floating point numbers, variable names with an appended percent sign % to represent integers, and variable names with an appended dollar sign $ to represent character strings. Occasionally, we shall use an appended at sign @ to stand for any one of these three possibilities (no suffix, %, or $). Thus, in the pseudocode we are defining, the names \( x \), \( \text{position} \), and \( \text{field} \) identify memory cells for floating point values, the names \( I\% \), \( \text{count}\% \), and \( \text{lower_limit}\% \) are valid names for cells storing integers, the names \( \text{word}$ \) and \( \text{first_name}$ \) are valid names for cells storing strings, and \( \text{item}@ \) identifies a quantity of any type.

The variables we have discussed so far are scalar or unstructured variables; each variable name stands for a single floating point number, integer, or character string. In scientific uses especially, we frequently want to refer to an aggregate of numbers (the three components of a position vector, the nine elements of a \( 3 \times 3 \) matrix, ...). To that end, the array is among the structured variables available in all scientific programming languages. An array may be one-dimensional (a vector), two-dimensional (an \( m \times n \) matrix), or even higher dimensional. As a structured entity, an array is identified by a single name, e.g., \( \text{data, values}\% \), or \( \text{names}$ \), depending on the data type of the elements of the array. Individual elements are identified by attaching an integer index or indices to the array name, e.g., \( \text{data}(I\%,J\%) \), \( \text{values}(4\%) \), or \( \text{names}$ (12\%) \). All by itself, the name of a scalar is sufficient to tell the compiler how much memory to reserve to store the scalar value. For arrays, however, we can’t simply use the name when it is first needed. In most languages, the source code must also convey how large the array will become during the execution of the program because the interpreter or compiler must set aside adequate memory to store the array before the interpretation or compilation can be completed. Thus, a program that uses arrays must include in its source code a directive informing the compiler or interpreter of the maximum dimensions of the array. In our generic discussion, we shall use a statement like

\[
\text{DIMENSION data}(4\%,251\%), \text{values}\%(8\%), \text{names}$ (25\%)
\]

to convey the number of elements in each array to the interpreter or compiler. From the beginning, be aware that some standard languages by default use array indices that run upwards from the value 1\% while others start the indices at the value 0\%. In both cases, the statement above creates an array \( \text{values}\% \), for example, with eight elements. In the first case, the indices run from 1\% to 8\% while in the second case they run from 0\% to 7\%.

In C and in FORTRAN 90 and perhaps in a few other languages, it is possible to change the dimensions of an array dynamically, i.e., to bypass the need for the size of arrays to be explicitly specified in a \( \text{DIMENSION} \) statement and, instead, arrange for the size of an array to be set, for example, in response to values input when the program is executed. The coding to accomplish this dynamic allocation of memory will be introduced if and when we find the need for it in the remainder of this book.
9.1.2 Essential Action Statements

The CPU must also be able to respond to a minimum collection of action statements to permit the assignment of values to variables, the performance of elementary arithmetic, and the transfer of information from and to a standard I/O device. We shall use the following special symbols and words for these operations:

- Assign a value to a named memory location:

  \[(\text{variable}) \leftarrow (\text{expression determining value to be assigned})\]

e.g.,

  \[\text{val} \leftarrow \sin(\pi \times x)\]

  \[\text{count} \% \leftarrow \text{count} \% + 1\%\]

  \[\text{first name} $\leftarrow "\text{David}"\]

  etc. Any valid arithmetic expression utilizing the symbols + for addition (or string concatenation), - for subtraction, * for multiplication, / for division, and ^ for exponentiation can appear on the right hand side of the assignment symbol $\leftarrow$. Further, most computer languages make available a wide variety of standard functions \((\sin, \cos, \atan, \sqrt{\text}, \exp, \ldots)\) to facilitate scientific computation.

- Obtain a value from the keyboard and store it in a (named) memory cell:

  \[\text{READ PROMPT="Enter first name: ", first name$}\]

- Transmit a value from a (named) memory cell and/or a (quoted) literal message to the screen:

  \[\text{WRITE "The value of the integral is ", value, "."}\]

We here recognize one further necessary refinement. In the above, we have simply requested information from the keyboard or directed output to the screen. Most programming languages will accept such simple statements, adopting a default format in which to expect the input or produce the output. Almost always, we will wish to exert some control over that format, and the standard languages provide means to give us that control. At base, we want to be able to dictate how many character positions should be occupied by the number, whether the number is to be output as an integer or as a decimal number, whether the decimal number is to be presented in scientific or conventional notation, how many digits are to be placed after the decimal point, whether the number is to be rounded, ... Further, we may want to create blank lines in the output, ignore blank lines in the input, or clear the screen. Since this list is (almost) endless, the task of describing the format in which input will be presented to the computer and specifying the format in which the computer should produce its output is among the most complicated tasks the programmer confronts. Unfortunately, different languages adopt significantly different ways to provide this facility. Hence, detailed discussion beyond this brief recognition of a need is best postponed until we discuss specific languages.

---

3 We confine our attention initially to data transfers from the keyboard and to the screen, postponing until later a discussion of the use of files.

4 We declare that integer constants shall be explicitly identified with a trailing percent sign and that string constants shall be enclosed in double quotation marks. Some languages use single quotation marks, while others will accept either (provided they are paired).
9.1 COMPONENTS OF A PROGRAMMING LANGUAGE

Table 9.1: Simple logical expressions and the circumstances under which each is true. Remember that we have introduced the suffix @ to indicate any of the standard data types.

⟨condition⟩ true if
- A@ = B@ A@ is the same as B@ (A@ equals B@)
- A@ < B@ A@ occurs before B@ in some collating sequence (e.g., numerical order, alphabetical order) (A@ less than B@)
- A@ > B@ A@ occurs after B@ in that collating sequence (A@ greater than B@)
- A@ < B@ A@ occurs after or is the same as B@ (A@ not equal to B@)
- A@ >= B@ A@ occurs after or is the same as B@ (A@ greater than or equal to B@)

Table 9.2: Truth tables for OR and AND.

<table>
<thead>
<tr>
<th>Proposition</th>
<th>Truth Value</th>
<th>Proposition</th>
<th>Truth Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>true OR true</td>
<td>true</td>
<td>true AND true</td>
<td>true</td>
</tr>
<tr>
<td>true OR false</td>
<td>true</td>
<td>true AND false</td>
<td>false</td>
</tr>
<tr>
<td>false OR true</td>
<td>true</td>
<td>false AND true</td>
<td>false</td>
</tr>
<tr>
<td>false OR false</td>
<td>false</td>
<td>false AND false</td>
<td>false</td>
</tr>
</tbody>
</table>

9.1.3 Logical Expressions and Conditions

Frequently, the CPU will need to make a decision on the basis of currently available information, performing different tasks depending on that decision. Usually, these decisions are binary. At base, the computer decides whether a particular logical proposition is true or false. Further, the propositions so examined are usually cast as a comparison of two quantities to determine how they would be ordered in some standard ordering sequence (numeric, alphabetic, ...). The six simple comparisons we might want to code and the circumstances under which the computer will judge each to be true are enumerated in Table 9.1.

Two additional capabilities are standard in computer languages. First, we endow our CPU with the ability to interpret compound conditions constructed out of simpler conditions either with the operator OR or the operator AND. A truth table showing the result of connecting each possible logical value with each of these operators is presented in Table 9.2. Second, we introduce the ability to negate a condition with the operator NOT, i.e., we introduce the expression

NOT ⟨condition⟩

which we define to be true if ⟨condition⟩ is false and false if ⟨condition⟩ is true.

Further (and finally), we recognize that we may occasionally find a need for a variable—a Boolean variable—that can assume only the two values true and false. Recognizing that different languages treat these variables differently, let us symbolize such a variable generically with a suffix ?.

EQUAL? ← N@ = M@

in response to which the computer will assess whether N@ is equal to M@ and set EQUAL? to true or false, depending on the outcome of that assessment. With this expansion of our language, conditions might be expressed not only by the comparisons illustrated in Table 9.1 but also by the
simple assertion of a single Boolean variable or by a logical expression involving combinations of comparisons and/or assertions of Boolean variables. Note, incidentally, that the statement in this paragraph reveals why computer languages must have a different symbol for assignment than for testing equality.\(^5\)

### 9.1.4 Syntactic Wrinkles

In some programming contexts, we will find it convenient—and sometimes even necessary—to convey additional information about the *structure* of the code to the CPU. We might wish:

- to place more than one statement on a single physical line. In our generic code, we shall use the character ‘;’ to separate individual statements on a single line. The three lines of code early in Section 9.1.2 might be combined in one line with the coding
  
  \[
  \text{val} \leftarrow \sin(\pi x); \quad \text{count}\% \leftarrow \text{count}\% + 1\%; \quad \text{first\_name$} \leftarrow \text{"David"}
  \]

- to spread a single statement over more than one physical line. In our generic code, we shall use the symbol → at the *end* of a line to indicate that the statement continues on the next line. Thus, for example, the two-line statement
  
  \[
  U(I\%, J\%) \leftarrow 0.25 \times (U(I\%+1\%, J\%) + U(I\%-1\%, J\%) \rightarrow + U(I\%, J\%+1\%) + U(I\%, J\%-1\%)
  \]

  is to be seen as a single statement.\(^6\)

- to group several statements together to form a *block* of statements so that they can be properly treated when the block is placed in a context in which the interpreter or compiler is expecting a single statement. In our generic code, we shall use the keywords BEGIN\_BLOCK and END\_BLOCK to “bracket” the several statements that we wish the compiler or interpreter to see as a single (compound) statement. For example, the coding
  
  ```
  \text{BEGIN\_BLOCK}
  \text{val} \leftarrow \sin(\pi x)
  \text{count}\% \leftarrow \text{count}\% + 1\%
  \text{first\_name$} \leftarrow \text{"David"}
  \text{END\_BLOCK}
  ```

  would group the three statements as a unit.

Different languages differ significantly in the way these three situations are coded. We shall be more explicit in subsequent sections as we address each language in turn.

### 9.1.5 Essential Control Structures

Four main control structures\(^7\) are generally provided in standard computer languages and figure significantly in the smooth expression of algorithms:

- **Sequence**, which is conveyed by the order of the statements in the algorithm.

---

\(^5\)We have used ← and =. In later sections, we will identify the symbols used in other languages.

\(^6\)Remember that spaces not in quoted strings are ignored by the compiler.

\(^7\)Since the CASE structure can be expressed in terms of nested IF-THEN-ELSE structures and the IF-THEN-ELSE structure is simply a CASE structure with only two cases, there are really only three fundamental structures (sequence, selection, and repetition). General theorems in computer science prove that no task will *require* more than these few control structures.
9.1. COMPONENTS OF A PROGRAMMING LANGUAGE

• Two-way selection, expressed generically with a statement like\(^8,9\)

\[
\text{IF} \langle \text{condition} \rangle \\
\quad \text{THEN} \langle \text{block1 of statements} \rangle \quad ! \text{the THEN clause} \\
\quad \text{ELSE} \langle \text{block2 of statements} \rangle \quad ! \text{the ELSE clause} \\
\text{END_IF}
\]

Here \langle block1 of statements \rangle is executed when \langle condition \rangle is true and \langle block2 of statements \rangle is executed when \langle condition \rangle is false. The ELSE clause may be omitted altogether if \langle block2 of statements \rangle is null. The flow diagram in Fig. 9.1(a) is often used to convey this structure.

• Multi-way selection\(^10\)

\[
\text{CASE} \\
\quad \text{OF} \langle \text{condition1} \rangle \quad \text{DO} \langle \text{block1 of statements} \rangle \\
\quad \text{OF} \langle \text{condition2} \rangle \quad \text{DO} \langle \text{block2 of statements} \rangle \\
\quad \text{OF} \langle \text{condition3} \rangle \quad \text{DO} \langle \text{block3 of statements} \rangle \\
\quad \vdots \\
\quad \text{OF OTHERS} \quad \text{DO} \langle \text{blockO of statements} \rangle \\
\text{END_CASE}
\]

In executing this overall statement, the computer will test each condition in turn, execute the block of statements associated with the first true condition it encounters, and then jump out of the CASE structure.\(^11\) The OTHERS clause may be omitted altogether if \langle blockO of statements \rangle is null. The flow diagram in Fig. 9.1(b) is often used to convey this structure.

• Repetition

\[
\text{LOOP} \\
\quad \langle \text{block1 of statements} \rangle \\
\quad \text{EXIT_LOOP WHEN} \langle \text{condition} \rangle \\
\quad \langle \text{block2 of statements} \rangle \\
\text{END_LOOP}
\]

Here, the statements in the body of the loop are executed repeatedly until \langle condition \rangle, which is tested at the indicated point in the loop, becomes true. Looping will continue forever unless repeated execution of the statements eventually causes \langle condition \rangle to become true. The flow diagram in Fig. 9.1(c) is often used to convey this structure.

In some more recent languages, the explicit construction of a loop is in some cases not necessary. For example, suppose we have a vector \(X\) containing \(N\) elements and we wish to create a second vector \(Y\), each of whose elements is—say—the sine of the corresponding element in \(X\). In the coding we have so far described, we would achieve this objective with an explicit loop involving the statements

\[
\text{CNT} \leftarrow 0 \quad ! \text{Set CNT to 0}
\]

for example, might as well be uncommented, since the comment really says no more than can be inferred from the statement itself.

\(^8\)The structure of this single statement is sufficiently distinctive—it is not complete until the keyword END_IF is encountered—that we need not use the symbol → described in Section 9.1.4 to indicate that the statement is spread over several physical lines.

\(^9\)We use the exclamation point to introduce comments. Judicious use of such comments, irrelevant though they may be to the computer, can clarify the algorithm for the human reader of the listing. To be useful, a comment should amplify the meaning of, or clarify the role of, the commented statement. The statement

\[
\text{CNT} \leftarrow 0 \quad ! \text{Set CNT to 0}
\]

for example, might as well be uncommented, since the comment really says no more than can be inferred from the statement itself.

\(^10\)As with the previous structure, the structure of this single statement is sufficiently distinctive—it is not complete until the keyword END_CASE is encountered—that we need not use the symbol → described in Section 9.1.4 to indicate that the statement is spread over several physical lines.

\(^11\)Not all compilers adopt this strategy. Some will execute the blocks associated with every true condition in the structure.
I% ← 0%
LOOP
    I% ← I% + 1%
    Y(I%) ← sin( X(I%) )
EXIT_LOOP WHEN I%=N%
END_LOOP

(We assume that the vectors have already been appropriately dimensioned.) Were we working with a language like IDL or MATLAB, which have built in array processing capabilities, this loop would be automatically constructed in response to the single statement

Y ← sin( X )

These languages simply understand that, when functions like the sine are supplied with an argument that is an array (whatever its dimension), the program is to compute an array of the same dimension, each of whose elements is that function of the corresponding element in the argument. These languages greatly simplify the coding of many operations involving arrays and, in addition, almost certainly generate a more efficient execution of the entire task. When available in the language in use, these capabilities should be exploited as much as possible.

Since blocks of statements may themselves involve any of these structures, the elementary structures can give rise to more complicated structures in which elementary structures are nested two or more deep. Note that, however complicated the THEN and ELSE clauses, the statements between CASE and END_CASE, or the body of the loop, the (outer) IF-END_IF statement, the (outer) CASE-END_CASE statement, and the loop as a whole are all seen by the compiler or interpreter logically as a single statement.

Although the LOOP structure identified above is quite general and, by itself, is adequate to accommodate all situations that might arise, structures that implement the terminating test in the middle of the loop are rare in actual computer languages. Three alternative versions are usually provided. In the first version, the test is at the beginning of the loop, e.g.,

WHILE ⟨condition⟩ DO (block of statements)

and the block is not executed at all if ⟨condition⟩ is false when the loop is entered. In the second version, the test is at the end of the loop, e.g.,

REPEAT (block of statements) UNTIL ⟨condition⟩

and the block will be executed at least once, whether ⟨condition⟩ is true or false when the loop is entered. Both of these versions will lead to infinite loops unless the statements in the body of the loop ultimately toggle the condition to the value that will terminate the loop. Flow diagrams depicting these structures are shown in Figs. 9.2(a) and (b).

The third version of a loop incorporates a built-in incrementation of an integer index and the automatic testing of that index, e.g.,

FOR I% ← IMIN% THRU IMAX% DO (block of statements)

When this loop is executed, the body of the loop is executed for I% having the value IMIN%, then for I% having the value IMIN%+1%, then for I% having the value IMIN%+2%, and ..., continuing until the loop has been executed for the largest value of I% that does not exceed IMAX%. The more sophisticated form

FOR I% ← IMIN% THRU IMAX% STEP INC% DO (block of statements)
Figure 9.1: Flow diagrams for the basic control structures: (a) two-way selection, (b) multi-way selection, (c) loop. Here, T, F, B, and C abbreviate true, false, block, and condition, respectively.

Figure 9.2: Flow diagrams for different loop structures: (a) WHILE-DO, (b) REPEAT-UNTIL, (c) FOR-DO. Again, T, F, B, and C abbreviate true, false, block, and condition, respectively.

gives the user control over the increment by which the index is increased before each new pass through the body of the loop. A flow diagram for this loop structure is shown in Fig. 9.2(c). Note that the loop we have here depicted will be executed once if IMIN% equals IMAX% and not at all if IMIN% exceeds IMAX%—but beware; different languages may behave differently in this regard.

Though it is not essential, one further control structure is standard in all scientific programming languages. Conveyed in our generic code by the statement

EXECUTE ⟨procedure name⟩
this single statement makes possible the invocation of a (properly defined) procedure in a program
(or for that matter in another procedure); it both facilitates a modular approach to the design of
programs and provides a means to avoid duplication of program modules that must be invoked more
than once to specify a task completely.

9.2 Sample Short Algorithms

Large algorithms for accomplishing complex tasks are frequently constructed by combining various
smaller algorithms. In this section, we enumerate several program fragments that may serve as
building blocks for the construction of larger algorithms.

1. Exchange the values stored in A@ and B@:\footnote{Note again the use of the character ! to flag comments. See footnote 9.}

\[
\begin{align*}
\text{TEMP}@ & \leftarrow \text{A}@ & ! \text{Save value in A}@ \\
\text{A}@ & \leftarrow \text{B}@ & ! \text{Copy B}@ \text{ to A}@ \\
\text{B}@ & \leftarrow \text{TEMP}@ & ! \text{Copy original A}@ \text{ from TEMP}@ \\
\end{align*}
\]

The temporary memory cell TEMP@ saves the original value of A@ so that it is still retrievable
after the value of B@ has been copied into cell A@, overwriting the original contents of cell A@.

2. Stopping a loop with a sentinel [and counting]:

\[
\begin{align*}
\text{SENTINEL}@ & \leftarrow \langle \text{agreed-upon special value} \rangle \\
\text{COUNT} & \leftarrow 0 \\
\text{LOOP} & \\
\text{READ ITEM}@ & \\
\text{EXIT_LOOP WHEN ITEM}@ = \text{SENTINEL}@ & \langle \text{Statements processing ITEM}@ \rangle \\
\text{COUNT} & \leftarrow \text{COUNT} + 1 \\
\end{align*}
\]

If the statements enclosed in [...] are included, then when the loop is completed the variable
COUNT will have as its value the number of values of ITEM processed. Note that the position
in the sequence at which COUNT is incremented is critical. As a general rule, counters should
be started at the value zero before anything happens and incremented by one immediately
after each of the events to be counted. More often than not, the end result of thoughtless or
unsystematic positioning of the incrementation will be a final count that is off by one, one way
or the other.

3. Stopping a loop by counting up:

\[
\begin{align*}
\text{NUMBER}_\text{OF\_TIMES} & \leftarrow \langle \text{desired number of executions} \rangle \\
\text{COUNTER} & \leftarrow 0 \\
\text{LOOP} & \\
\text{EXIT_LOOP WHEN COUNTER} = \text{NUMBER}_\text{OF\_TIMES} & \langle \text{block of statements} \rangle \\
\text{COUNTER} & = \text{COUNTER} + 1 \\
\end{align*}
\]

Again, careful initialization of the counter and careful positioning of its incrementation are
critical to avoiding off-by-one errors.

4. Stopping a loop by counting down:
TIMES_REMAINING% ← (desired number of executions)
LOOP
   EXIT_LOOP WHEN TIMES_REMAINING% = 0%
   (block of statements)
   TIMES_REMAINING% ←− TIMES_REMAINING% - 1%
END_LOOP

This coding has a small advantage over the previous coding because it requires only one variable (TIMES_REMAINING) to control the loop. Note, however, that the initial value of that variable is irrecoverably lost by the time execution of the loop has been completed.

5. Summing [and counting]:

SENTINEL ← (agreed-upon special value)
SUM ← 0.0
[COUNT% ←− 0%]
LOOP
   READ ITEM
   EXIT_LOOP WHEN ITEM = SENTINEL
   SUM ←− SUM + ITEM
   [COUNT% ←− COUNT% + 1%]
END_LOOP
WRITE "The sum is "; SUM

This algorithm for adding numbers involves the same steps you would use to accomplish the task on a pocket calculator: initialize the accumulator to 0.0, enter each new value in turn, push the ‘add’ button after each entry, stop after the last value has been processed, and read the final value in the accumulator. Note the explicit decimal point in the floating point constant 0.0. Since some compilers and interpreters in some circumstances will treat numerical constants without explicit decimal points as integers, possibly producing unintended results, prudence dictates habitually placing an explicit decimal point in all integer constants that are in fact to be treated as floating point values. In the present situation, the variable SUM is implicitly declared to be a floating point variable, so the statement SUM ←− 0 would result in an internal conversion of the integer value 0 to the floating point value 0.0, but it nonetheless pays to be cautious.

6. Finding extreme, stopping with a sentinel:

SENTINEL@ ←− (agreed-upon special value)
READ (first) ITEM@ from list
EXTREME@ ←− ITEM@
LOOP
   READ (next) ITEM@ from list
   EXIT_LOOP WHEN ITEM@ = SENTINEL@
   IF ITEM@ and EXTREME@ are out of order
      THEN EXTREME@ ←− ITEM@
   END_IF
END_LOOP
WRITE "The extreme value is "; EXTREME@

While this procedure can be executed with no a priori knowledge of the number of items and no special assumptions about the list to come, it has the disadvantage of requiring the first item of the list to be treated differently from the subsequent items. [Questions to the reader: (1) Does this procedure behave sensibly if the sentinel is entered as the first item? How might

\[13\] For example, 5/2 = 2 in integer arithmetic; 5.0/2.0 = 2.5 in floating point arithmetic.
you improve the procedure on that score? Should you bother? (2) What error would occur in the output if the `EXIT_LOOP WHEN` statement were placed just before the `END_LOOP` statement?]

7. Finding extreme, stopping by counting:

```plaintext
NUMBER_OF_VALUES% ← (number of items to be presented)
READ (first) ITEM@ from list
COUNTER% ← 1% ! Count item
EXTREME@ ← ITEM@ ! Assume first is extreme
LOOP
   EXIT_LOOP WHEN COUNTER% = NUMBER_OF_VALUES%
   READ (next) ITEM@ from list
   COUNTER% ← COUNTER% + 1%
   IF ITEM@ and EXTREME@ are out of order
      THEN EXTREME@ ← ITEM@
   END_IF
END_LOOP
WRITE "The extreme value is"; EXTREME@
```

[Question to the reader: What error would occur in the report if `COUNTER%` were incremented just before the `EXIT_LOOP WHEN` statement?]

8. Sequential search:

```plaintext
ITEM_Sought@ ← ⟨item to be found⟩
NO_ELEMENTS% ← ⟨number of items in ITEMS@⟩
POINT% ← 0% ! Start at beginning
LOOP
   POINT% ← POINT% + 1% ! Advance pointer to next item
   EXIT_LOOP WHEN ITEMS@(POINT%) = ITEM_Sought@ OR POINT% = NO_ELEMENTS%
END_LOOP
```

We assume that, by the time this procedure is invoked, the list of items to be examined has been stored in the one-dimensional array `ITEMS` and that we know the number of elements in that array. Note that the pointer `POINT%` is stepped through the records one at a time as the search unfolds. Finally, when this loop terminates, the item sought has been found if and only if the value in the record identified by `POINT%` matches the item sought.

### 9.3 Two Larger Algorithms

In this section, we discuss two algorithms in some detail, partly to illustrate the general features described in Section 9.1 more fully and partly to lay out the algorithms generically before we implement them in specific languages.

#### 9.3.1 Solving Laplace’s Equation

Among the more important equations in mathematical physics, Laplace’s equation appears in the study of electromagnetic fields, steady state heat flow, fluid mechanics, and many other contexts. In two-dimensions and in Cartesian coordinates $(x, y)$, the equation assumes the form

$$\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} = 0 \quad (9.1)$$
9.3. TWO LARGER ALGORITHMS

Figure 9.3: The geometry for a simple problem involving Laplace’s equation.

for a function $U(x, y)$, which may be interpreted as an electrostatic potential, a temperature distribution, a velocity potential describing the incompressible, steady-state flow of a fluid, ... Beyond the partial differential equation itself, a complete problem requires the statement of boundary conditions—often the stipulation of the value of $U$—at all points on the boundary of the region in which a solution is sought. Thus, for example, a complete problem might seek a solution to the Laplace equation in a square, subject to the requirement that the solution assume the value 0.0 on three edges of the square and the value 100.0 on the fourth edge, as shown in Fig. 9.3(a). Physically, this solution would convey the temperature within the square when three of its edges are maintained at 0° and the fourth edge is maintained at 100° or the electrostatic potential within the square when three edges are maintained at a potential of 0 Volts and the fourth edge is maintained at a potential of 100 Volts.

The basis for a simple algorithm for solving this problem numerically involves imposing an $N \times N$ regular grid of points $(x_i, y_j)$, $1 \leq i, j \leq N$, on the region. Then, we declare that we have found a solution when we know appropriate values for $U_{i,j} = U(x_i, y_j)$ at each grid point. The simplest boundary conditions, of course, tell us the values at all points for which $i$ and/or $j$ is either 1 or $N$. The values we seek for the other points ($2 \leq i, j \leq N - 1$) must in some sense reflect the differential equation. If, however, we know the values of $U$ at three consecutive points along a line parallel to the $x$ axis, say, we can use the approximation (see exercises)

$$
\frac{\partial^2 U}{\partial x^2} \approx \frac{U_{i+1,j} - 2U_{i,j} + U_{i-1,j}}{\Delta x^2}
$$

where $\Delta x$ is the (constant) spacing of consecutive grid points. A similar expression applies as an approximation to the second derivative of $U$ with respect to $y$. Discretizing Eq. (9.1) by substituting these finite-difference approximations for the derivatives and then rearranging the resulting equation, we conclude that

$$
U_{i,j} \approx \frac{1}{4} \left( U_{i+1,j} + U_{i-1,j} + U_{i,j+1} + U_{i,j-1} \right)
$$

i.e., that the value we should assign to the “squared” point in Fig. 9.3(b) is the average of the values we assign to its four nearest neighbors (the “circled” points).

We, of course, have one equation like Eq. (9.3) for each of the interior points in the illustrated grid. We also have exactly as many unknowns as we have interior points. Further, the equations are linear. Thus, we have deduced a (probably large) set of linear equations to be solved simultaneously for the unknown values $U_{i,j}$ at the interior points. For this set of equations, a suitable strategy—called relaxation— involves guessing a starting solution and then refining that solution iteratively by stepping systematically and repeatedly through the grid of interior points, replacing the value at
each point by the average of the values at its four nearest neighbors. Each pass through the entire grid constitutes one iteration, and we keep going—iteration after iteration—until we are satisfied that the process has converged satisfactorily.\(^{14}\) Supposing that we seek a solution on a \(15 \times 15\) grid of points, we make a first pass at constructing an algorithm to implement this overall strategy by listing the statements

```
DIMENSION U(15%,15%)
Set U(I%,J%) ← 0.0 for all values of I%, J%
Set U(15%,J%) ← 100.0 for all values of J%
FOR ITCNT% ← 1% THRU 30% DO Conduct one iteration
Write solution to output device
```

Here, we (1) reserve adequate space for the necessary array, (2) establish a starting solution in that array, (3) set the boundary conditions, (4) conduct 30 iterations, and (5) display the solution.

To refine the crude statements in the above algorithm, we would have to expand the implied loops. Indeed, since we are working with a two-dimensional array, stepping through all elements of the array (either in assigning initial values or in iterating once through the array) will require a double loop. Thus, we might express this algorithm more explicitly as

```
DIMENSION U(15%,15%)
FOR I% ← 1% THRU 15% DO
    FOR J% ← 1% THRU 15% DO U(I%,J%) ← 0.0
FOR J% ← 1% THRU 15% DO U(15%,J%) ← 100.0
FOR ITCNT% ← 1% THRU 30% DO Conduct one iteration
    FOR I% ← 2% THRU 14% DO
        FOR J% ← 2% THRU 14% DO
            U(I%,J%) ← 0.25 * ( U(I%+1%, J%) + U(I%-1%, J%) + U(I%, J%+1%) + U(I%, J%-1%) )
    FOR I% ← 1% THRU 15% DO
        FOR J% ← 1% THRU 15% DO WRITE U(I%,J%)
```

We shall refine this algorithm even further as, later in this chapter, we implement it in various computer languages.

### 9.3.2 File Output/Input

The most convenient and flexible way to make a permanent record of numerical data is to store it in an ASCII file. Among other advantages, such files can be created in any number of ways, they can be examined with ordinary text editors, and they facilitate importing the data into whatever graphical package provides the best visualization of the specific features we wish to see. Most often the data are naturally organized into one or more two- or three-dimensional arrays, which frequently represent scalar or vector fields. In this section, we declare a standardized file format for the storage of these arrays and discuss a general algorithm by which files so structured can be created. In later sections, we describe how to implement this algorithm in various computer languages. Of course, the data file as an intermediary is unnecessary if generation and display occur in the same program. These files are especially useful when data generated in one program are to be transferred to another program or when a program-independent permanent record of the data is desired.

\(^{14}\)The most common criterion of convergence involves comparing values until no element in one iterate differs from its counterpart in the next iterate by more than some predetermined tolerance. Developing suitable (and reliable) criteria for determining when satisfactory convergence has been achieved, however, is difficult. Because we are here focusing on programming aspects rather than algorithmic refinements, we shall ignore the issue of convergence for the moment. We shall return to it at appropriate points in later chapters.
Table 9.3: Sample data file having the structure described in Section 9.3.2. The line numbers in the first column and the comments in the third column of this table are not in the actual file.

<table>
<thead>
<tr>
<th>Line Number</th>
<th>Line in file</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>01:</td>
<td>Example Data File; Author: DMC; Date: 7-21-01</td>
<td>ID of file</td>
</tr>
<tr>
<td>02:</td>
<td>First line of comments.</td>
<td></td>
</tr>
<tr>
<td>03:</td>
<td>Second line of comments.</td>
<td></td>
</tr>
<tr>
<td>04:</td>
<td>Third line of comments.</td>
<td></td>
</tr>
<tr>
<td>05:</td>
<td>Fourth line of comments.</td>
<td></td>
</tr>
<tr>
<td>06:</td>
<td>1</td>
<td>n (number of arrays)</td>
</tr>
<tr>
<td>07:</td>
<td>2</td>
<td>$n_x$</td>
</tr>
<tr>
<td>08:</td>
<td>3</td>
<td>$n_y$</td>
</tr>
<tr>
<td>09:</td>
<td>1</td>
<td>$n_z$</td>
</tr>
<tr>
<td>10:</td>
<td>0.33</td>
<td>$A_{111}$</td>
</tr>
<tr>
<td>11:</td>
<td>0.28</td>
<td>$A_{211}$</td>
</tr>
<tr>
<td>12:</td>
<td>0.50</td>
<td>$A_{121}$</td>
</tr>
<tr>
<td>13:</td>
<td>0.42</td>
<td>$A_{221}$</td>
</tr>
<tr>
<td>14:</td>
<td>0.66</td>
<td>$A_{131}$</td>
</tr>
<tr>
<td>15:</td>
<td>0.57</td>
<td>$A_{231}$</td>
</tr>
</tbody>
</table>

At base, we imagine that the data files we create will store some number $n$ of identically dimensioned three-dimensional arrays $A_{ijk}$ whose three indices will assume $n_x$, $n_y$, and $n_z$ values, respectively, though at times $n_z$ will have the value 1 so that the storage of two-dimensional arrays can be accommodated. We adopt the following standard format for all data files:

- five lines of comments, possibly including title, author, and date;
- one line specifying the number $n$ of arrays in the file;
- one line specifying the number $n_x$ of values assumed by the first index $i$ of $A_{ijk}$;
- one line specifying the number $n_y$ of values assumed by the second index $j$ of $A_{ijk}$;
- one line specifying the number $n_z$ of values assumed by the third index $k$ of $A_{ijk}$; and finally
- $n_xn_yn_z$ lines specifying the values in the first array—one value per line in the order that results from the generic coding

\[
\begin{align*}
\text{FOR } k\% & \leftarrow 1\% \text{ THRU } NZ\% \text{ DO} \\
& \quad \text{FOR } j\% \leftarrow 1\% \text{ THRU } NY\% \text{ DO} \\
& \quad \quad \text{FOR } i\% \leftarrow 1\% \text{ THRU } NX\% \text{ DO WRITE } A(i\%, j\%, k\%) \\
& \quad \text{—followed by a similarly structured presentation of the values in the second array, the third array, etc. That is, the order of elements for each array is created by allowing the first index to vary the most rapidly, the second index to vary next rapidly, and the third index to vary least rapidly.}
\end{align*}
\]

The data file shown in Table 9.3 is a simple example. This file contains one $2 \times 3 \times 1$ (2 rows $\times$ 3 columns $\times$ 1 plane) array, namely

\[
A = \begin{pmatrix} A_{111} & A_{121} & A_{131} \\ A_{211} & A_{221} & A_{231} \end{pmatrix} = \begin{pmatrix} 0.33 & 0.50 & 0.66 \\ 0.28 & 0.42 & 0.57 \end{pmatrix}
\]

with the elements in the first column, then the elements in the second column, and finally the elements in the third column recorded in the file.

\[^{15}\text{In some languages, the actual values of these indices will range from 0 to } n_\ast - 1; \text{ in other languages, the values will range from 1 to } n_\ast. \text{ (The subscript } \ast \text{ stands for } x, y, \text{ or } z. \text{ In a few languages, the user has control over the range of the indices.}\]
To be even more explicit, we would have to surround the basic output statement presented above with statements that dimension arrays appropriately, calculate (or otherwise determine) the values to insert in the arrays, prepare the file for access, and close the file after the last datum has been written to the file. To illustrate, suppose we seek to explore the two-dimensional scalar field given in the $xy$ plane by the function $f(x, y)$. Then, generating the desired file involves two main steps: (1) creation of an internal array containing values of the function at a grid of points overlayed on the region of interest in the $xy$ plane and (2) writing those values into a suitably labeled file structured as described earlier in this section. Though it may take a bit of exploration to decide on suitable ranges for the independent variable, let us here decide to examine the function in the region defined by $x_{\min} \leq x \leq x_{\max}$ and $y_{\min} \leq y \leq y_{\max}$ and to divide the $x$ interval with $n_x$ grid points into $n_x - 1$ segments and the $y$ interval with $n_y$ grid points into $n_y - 1$ segments. First, we prepare variables and arrays with the statements

```
DIMENSION ARRAY(n_x, n_y)
NARR% ← 1%
NX% ← n_x
NY% ← n_y
NZ% ← 1%
DX ← (x_{\max} - x_{\min}) / (n_x - 1)
DY ← (y_{\max} - y_{\min}) / (n_y - 1)
```

! Prepare array for values
! Set number of arrays
! Set number of grid points
! in each coordinate

Then, we evaluate the function $f(x, y)$ with the double loop expressed in the code

```
FOR J% ← 1% THRU NY% DO BEGIN_BLOCK
  YF ← y_{\min} + (J% - 1%) * DY
  FOR I% ← 1% THRU NX% DO BEGIN_BLOCK
    XF ← x_{\min} + (I% - 1%) * DX
    ARRAY(I%, J%) ← f(XF, YF)
  END_BLOCK
END_BLOCK
```

Finally, we establish communication between the file and the program (in the jargon, we attach the file to the program on a selected channel), write the necessary labels and values to the file, and disconnect (i.e., detach the file from the program with the statements\textsuperscript{16}

```
ATTACH FILE ⟨filename⟩ ON CHANNEL 1 FOR WRITING
WRITE TO CHANNEL 1, "Line of explanation (Title, Author, Date?)"
WRITE TO CHANNEL 1, "Line of explanation"
WRITE TO CHANNEL 1, "***"
WRITE TO CHANNEL 1, "***"
WRITE TO CHANNEL 1, NARR%
WRITE TO CHANNEL 1, NX%
WRITE TO CHANNEL 1, NY%
WRITE TO CHANNEL 1, NZ%
FOR J% ← 1% TO NY% DO FOR I% ← 1% TO NX% DO WRITE TO CHANNEL 1, ARRAY(I%, J%)
CLOSE FILE ON CHANNEL 1
```

Here, the first statement identifies the file and assigns to it a number—any number will do\textsuperscript{17}—to be used for subsequent reference to the file. Next, the first five WRITE statements send five lines of

\textsuperscript{16} A new file will be created, overwriting any existing file by the specified name. Some programming languages may provide a warning if an existing file will be deleted.

\textsuperscript{17} Be aware that some languages reserve a few channel numbers for special "files", e.g., the keyboard or the screen.
comments, all five of which must be physically present even if fewer than five are needed to contain necessary information about the file. Then, the sixth WRITE statement writes the number of arrays to the file, the seventh and eighth write the x and y dimensions to the file, and the ninth writes the z dimension to the file. Finally, the double loop writes the elements of the array to the file in the proper order, and the last statement makes sure the file is properly detached from the program so that it can be accessed by other programs.

Instead of writing data to a file, we frequently will need to read data from a file, perhaps as a means to import data generated by one program for use in a different program. Slight modification of the ATTACH and READ statements introduced earlier provide for that action. Specifically, the statements

ATTACH FILE (filename) ON CHANNEL 1 FOR READING
READ FROM CHANNEL 1, Comma-separated list of variables for storage of values

CLOSE FILE ON CHANNEL 1

Here, the first statement opens an existing file for reading (and displays an error if the specified file does not exist), the remaining statements save the last read the information in the file and stores it in the specified variables, and the last statement detaches the file when, presumably all data have been read. Note that the structure of the READ statements must reflect the structure of the file, and the variables specified must have the data types of the values to be read. Normally, the READ statement reads a line at a time, but in some languages each READ statement reads as many values as needed to complete the list of variables, however these values are distributed over lines.

In the remaining sections of this chapter, we shall implement the two algorithms described in this section in different languages and, especially for the second algorithm, in a variety of different contexts.

9.6 Solving Laplace’s Equation with C

Note: All C programs (*.c) and C-created data files (*.c.dat) referred to in this chapter are available in the directory $HEAD/cc, where (as defined in the Local Guide) $HEAD must be replaced by the appropriate path for your site.

Rather than systematically laying out features of C as a programming language before presenting a program, we elect to present a simple program illustrating several of those features and then comment thereon. A C program implementing the algorithm laid out in Section 9.3.1 for solving Laplace’s equation is presented in Table 9.4. Note the following about this program:

- C is sensitive to case.
- Comments are introduced with the characters /* and terminated with the characters */.
- Directives to the compiler are introduced with the character #. Only a very small set of built-in routines is included without explicit request. If we want to do any I/O at all, we must explicitly include the standard I/O header file stdio.h with the first include statement. Further, using even simple mathematical operations requires explicit inclusion of the math header file math.h.
- The compiler directives beginning define provide a means to assign character strings to symbols such that subsequent use of the symbol triggers a literal replacement of the symbol with the assigned character string. These symbols are not variables, since they cannot be assigned different values during execution of the program. This compiler directive provides a way to bring together at the beginning of a program all those quantities whose value may change from time to time and place them at a point where they are easy to find in the listing so that changing the value means editing the file in only one place.
/* PROGRAM laplace.c */

/* This program solves Laplace’s equation in a square when
three sides of the square are maintained at zero potential
and the fourth side is maintained at a potential of 100 V.
The solution on a 15 x 15 grid is stored in the array U.  */

#include <stdio.h>
#include <math.h>
#define xdim 15
#define ydim 15
#define maxit 300

main()
{
    /***** Declare variables *****/
    float U[xdim][ydim];   /* For solution */
    int i, j, itcnt;     /* For loop control */
    /***** Initialize U(I,J); set boundary conditions *****/
    for(i=0; i<xdim; i++)
        for(j=0; j<ydim; j++)
            U[i][j] = 0.0;
    for(j=0; j<ydim; j++)
        U[xdim-1][j] = 100.0;
    /***** Iterate to solution *****/
    for(itcnt=1; itcnt<=maxit; itcnt++)
        for(i=1; i<xdim-1; i++) /* Conduct one iteration */
            for(j=1; j<ydim-1; j++)
                U[i][j] = 0.25 * ( U[i+1][j] + U[i-1][j] + U[i][j+1] + U[i][j-1] );
    /***** Display solution *****/
    for(j=0; j<ydim; j++)
    {
        for(i=0; i<xdim; i++)
            printf( "\%7.2f", U[i][j] );
        printf( "\n" );
    }
• The main program is introduced with the statement `main()`.

• Every C statement must be terminated with a semicolon. Thus, statements can be freely spread over several lines. The compiler doesn’t take the statement to be complete until it encounters the semicolon. The semicolon can also be used to separate distinct statements placed physically on the same line.

• Legal variable names—the C term is `identifier`—begin with a letter, which may be followed by any number of characters chosen from the letters A–Z, the letters a–z, the numeric digits 0–9, and the underscore _, which is regarded as a letter (so it can also be a first character in an identifier). Though the feature varies among compilers, we can count on at least the first 31 characters being recognized.

• C uses explicit data typing, so all variables to be used must be declared as `float` or `int` (integer) or `char` (character) or . . . .

• C uses the symbols `+`, `−`, `∗`, and `/` for addition, subtraction, multiplication, and division.

• To raise a variable to a power in C, we must invoke the function `pow`, which has two arguments, the first identifying the variable to be raised to a power and the second specifying the power. For example, $x^2$ would, in C, be coded as `pow(x, 2.0)` Further, `pow` is not defined unless the header file `math.h` has been included.

• The indices of elements of arrays are enclosed in `square` brackets and each index has its own set of brackets. Further, the dimensioning of an array is incorporated in the statement that declares the variable. Array indices run upwards from 0 and an array declared, for example, as `float U[15][15]`; will have elements running from $U[0][0]$ to $U[14][14]$ with 15 values for each index.

• C uses the equal sign for assignment. The operators `>`, `>=`, `<`, `<=`, `==`, and `!=` are used in logical expressions to symbolize greater than, greater than or equal to, less than, less than or equal to, equal to, and not equal to, respectively; the operators `&&`, `||`, and `!` symbolize AND, OR, and NOT, respectively.

• A counted C loop is introduced with the keyword `for`, which is followed by three arguments enclosed in parentheses and separated by semicolons. The first of these arguments (e.g., `i=0`) identifies the loop index and initializes it, the second (e.g., `i<xdim`) specifies the condition which must be true for the loop to continue, and the third (e.g., `i++`, which is a shorthand for `i=i+1`) specifies (cryptically) that the index is to be incremented by one with each pass through the loop. The condition is tested at the beginning of the loop and the incrementation of the index takes place after the statements in the body of the loop have been executed. Thus, on the first pass through the loop, the statements are executed with the initial value of the index, and the loop will not be executed at all if the condition is false for the initial value of the index. Note that consecutive arguments are separated by a semicolon.

• The `printf` statement directs its output to the standard output device, which is the screen [and is always “attached”—C uses the verb “open”—and hence need not be explicitly attached and detached (i.e., opened and closed)].

• The first argument in the `printf` statement is a quoted string that conveys not only characters to be transmitted literally to the output device but also specifications effecting the formatting of the value(s) of the variable(s) that is (are) listed in the second and subsequent arguments. In particular, the notation `
` transmits the control character to start a new line.

• The format of the output produced by the `printf` statement is specified by inclusions in the opening character string, each of which is introduced by a percent sign `%`. The specification `%7.2f`, for example, stipulates that the number to be displayed should be displayed in ordinary decimal notation—the f—right-justified in a field (region) that is seven characters wide—the
7—with the number rounded to have two digits—the 2—after the decimal point. In this particular problem, we know that all values should be positive and no value will be larger than 100.18 Thus, with the specified format, the largest value will be displayed as 100.00, right-justified in a seven-character field. Since this number requires six character positions, there will be at least one space between consecutive entries in each line of output.19,20

- The printf statement that sends a single value of U to the screen is embedded in the inner loop in the output section of this program. Note that that loop runs 15 times and that no new-line character is included in the format string in that inner loop. The new-line character is supplied once in the outer loop for every complete execution of that inner loop. Thus, a new line will be started only after each group of fifteen values has been output by the inner loop.

- In C, opening and closing braces ({ and }) are used to group statements together into blocks. Note in particular that the entire program is enclosed within the opening brace in the line immediately following the phrase main() and the closing brace in the very last line of the listing. The one other block in this program is in the display section where two statements (a for loop and the statement printf("\n");) must be blocked together to form the “object” of the outer for loop.

- Note that, in the loop generating the output, for each j, i migrates through its several values. Thus, the rows in the output are labeled by values of j, i.e., of y, and the columns in the output are labeled by values of i, i.e., of x.

This program is stored in a file named laplace.c in the directory $HEAD/cc. Once that file has been copied to the default directory, it can be compiled, linked, and run with commands as described in the Local Guide. In UNIX, those commands might be

```c
cc -o laplace.xc laplace.c
./laplace.xc
```

In response to these commands to the operating system, the program will be executed and the output—15 lines of 15 numbers each—will be displayed on the screen.22

The solution displayed on the screen when the program is run is, of course, difficult to interpret (and would be even more so if we had solved the equation on a finer grid). Anticipating that we might want to import the output of this program into another—possibly a plotting—program, we might wish to direct the output to a file rather than to the screen. In UNIX and in a Windows command window, we could use the simple statement

```c
./laplace.xc > laplace_c.dat
```

---

18Solutions to Laplace’s equation can never have in the interior a value larger than the largest value nor smaller than the smallest value on the boundary.

19If we had to allow for the possibility of values ranging downward to -100, we would need one more character position (%8.2f) to provide the position for a possible minus sign and guarantee at least one space between any possible consecutive entries in each line.

20One alternative to the f format is d, which is used for integers, e.g., %8d for an integer to be right-justified in a field eight characters wide. Another alternative, used particularly when the size of floating-point numbers varies considerably or is unknown a priori, is e, which outputs floating point numbers in scientific notation, e.g., %15.6e for a floating point number to be right-justified in fields fifteen characters wide, presented with six digits after the decimal point (and formatted—depending on the compiler—either with one digit or a zero before the decimal point). The format %15.6e, for example, will present the number 100.00 as 0.100000e 03 or 1.000000e 02, requiring in both cases twelve character positions of the fifteen specified by the format. Further, note that the simpler specifications %f, %d, and %e will allow the compiler to make its own judgment about the appropriate field sizes.

21Throughout this book, we shall assume that we are working in a UNIX environment, though cc is fairly standard for invoking the C compiler. In other operating systems, the characters ./ may or may not be necessary to launch an executable program in the current directory. Further detail on how to invoke the C compiler and run an executable program in your environment will be described in the Local Guide.

22Incidentally, note that, if necessary, execution of a C program can usually be aborted by typing ⟨CONTROL/C⟩.
to achieve that end without having to edit and recompile the program at all. Alternatively, and more generally, we would have to replace the output portion of the program with code that opens a file, writes the output to that file, and then closes the file. We might, for example, edit `laplace.c` in two ways. First we would add the statement

```c
FILE *fptr; /* For file pointer */
```

to the declaration section, thereby identifying a variable to play the role of a file pointer. Second, we would replace the output section with the statements

```c
fptr=fopen( "laplace_c.dat", "w" );
for(j=0; j<ydim; j++)
{
    for(i=0; i<xdim; i++)
        fprintf( fptr, "%7.2f", U[i][j] );
    fprintf( fptr, "\n" );
}
fclose( fptr );
```

using `fopen` to open the file for writing (the "w") and to associate it with the pointer `fptr`, `fprintf` (rather than `printf`) to write everything to the file, and finally `fclose` to close the file. Except for the added first argument, the `fprintf` statement is structured in the same way as the `printf` statement. The program so edited is stored in the file `laplace_file.c` in the directory `$HEAD/cc`. Once copied to the default directory, it can be compiled, linked, and run as with `laplace.c` above (see the `Local Guide`). In the end, the output file `laplace_c.dat` will be created in the default directory.
9.7 Additional Features of C

The example program in the previous section has not provided a vehicle for illustrating a few additional important capabilities of the language. We here address those features.

9.7.1 Multi-Way Selection

The basic decision-making structure in C involves keywords, conditions, and actions in the general pattern conveyed by the listing

```
if ⟨condition1⟩
   ⟨block1 of statements⟩
else if ⟨condition2⟩
   ⟨block2 of statements⟩
   ...
else if ⟨conditionn⟩
   ⟨blockn of statements⟩
else
   ⟨catch-all block of statements⟩
```

Test ⟨condition1⟩. Execute ⟨block1⟩ and exit when ⟨condition1⟩ true.
Test ⟨condition2⟩ (if ⟨condition1⟩ false). Execute ⟨block2⟩ and exit when ⟨condition2⟩ true.
Test ⟨conditionn⟩ (if ⟨condition(n−1)⟩ false). Execute ⟨blockn⟩ and exit when ⟨conditionn⟩ true.
Execute ⟨catch-all block⟩ and exit when all conditions false.

In this construction, if a particular block of statements consists of a single statement, it can simply be presented in the proper position in the structure (terminated, of course, with a semicolon). When a block requires more than one logical statement, the statements as a group must be enclosed in braces so the compiler will treat them logically as a single statement. Further, two-way selection is simply a version of multi-way selection in which there is only one condition in this sequence, and the final else clause can be omitted if it contains no statements. The blocks can contain any number of statements, each terminated with a semicolon, and the conditions can be logical variables, single logical expressions, or compound logical expressions.

The above chain of if-else-if structures is a bit awkward. As an alternative, C also provides the structure

```
switch ⟨expression⟩
{
   case ⟨constant-expression1⟩: ⟨block1 of statements⟩
   case ⟨constant-expression2⟩: ⟨block2 of statements⟩
   :.
   case ⟨constant-expressionn⟩: ⟨blockn of statements⟩
   default: ⟨block-def of statements⟩
}
```

Here, the expression following the keyword switch must yield an integer value. When that integer matches the value of one of the constant expressions distinguishing a case, the statements in that case will be executed. If no case is satisfied, the statements in the default case will be executed, though the default case can be omitted altogether if specific actions are to be coded only for the cases explicitly mentioned.

9.7.2 Reading Data from Keyboard

Data can be read from the keyboard in C with a variety of statements. In the simplest approach, we might use the command printf to print a prompting message conveying the item wanted and then the command scanf to read that item from the keyboard and store it in a suitable variable with statements like
9.7. ADDITIONAL FEATURES OF C

printf( "\nSuitable prompting message: " );
scanf( "%f", &value );

(Note the ampersand preceding the variable name; it makes sure that the argument refers explicitly to the address of the memory cell(s) in which the value of the variable is to be stored and must be present for proper functioning.) Finally, note that, when coded like

scanf( "%f %d", &value, &index );

two values would be entered in response to the prompting message, the values being separated by one or more spaces or by one or more tabs. In this specific example, of course, the two variables would previously have been declared as a single-precision floating point value and an integer, respectively.

9.7.3 Additional Loops

Beyond the for loop, C makes available a while loop and a do/while loop. The while loop places the condition controlling the loop at the beginning and is coded with a structure like

```
while (condition)
  (block of statements)
```

where each statement in the block must—as always—be terminated with a semicolon and, except when it consists of a single statement, the block must be enclosed in braces. The statements in the block will be executed repeatedly as long as (condition) remains true, and the loop will be infinite unless the action of the statements in the loop ultimately turns the condition to false.

The do/while loop places the condition controlling the loop at the end and is coded with a structure like

```
  do
    (block of statements)
  while (condition);
```

where each statement in the block must—as always—be terminated with a semicolon and, except when it consists of a single statement, the block must be enclosed in braces. The statements in the block will be executed repeatedly as long as (condition) remains true, and the loop will be infinite unless the action of the statements in the loop ultimately turns the condition to false.

9.7.4 Functions and Global Variables

As with most computer languages, C also admits the possibility of creating new commands/functions through the use functions. In essence, a function is a self-contained segment of code that is given a name of its own and that can be invoked in another program with a single statement. In outline, we define a function with a structure like
dt FuncName( dt1 Arg1, dt2 Arg2, ...) 
{
  Statements coding the desired action.
}

Here, dt specifies the data type of the value returned by the function (and may be void if no value is returned); FuncName is a user-selected name for the function; and dt1, dt2, ... specify the data types of the arguments Arg1, Arg2, ..., which are the names of the variables that communicate (either by providing input to the function or providing values returned to the calling program) between the calling program and the function. All other variables in the function are invisible to the calling program (and can, in fact, have the same names as variables in the calling program without causing confusion). The statements defining the function must appear together in the listing of a full program and each function definition must appear in the listing before its first invocation within the total program unless the compiler is alerted to its later presence in the listing with a statement that simply repeats the first line of the function definition. Once in place, the function itself can be invoked at any point in the program with the single statement

FuncName( Arg1, Arg2, ... )

This time, the arguments can be variable names identifying the storage location for the values to be submitted to the function or they can be actual values. Further, while the order of the arguments is critical to the proper interpretation of each, the variable names used in invoking a function need not be—and in general will not be—the same as those used in defining the function in the first place. Data types and dimensions must match, but names need not.

If the data type of the function is not void, then somewhere—often at the very end—in the function definition there must appear a statement of the form

return (expression giving value to be returned)

Then, the function would be invoked with a statement of the form

VarName = FuncName( Arg1, Arg2, ... )

where VarName is the name of a variable of the appropriate data type into which the value returned by the function is to be stored.

In later chapters, we will at times find it necessary to “sneak” values of some variables into a function without using the “official” arguments. To do so in C, we must make use of global variables, which are declared by placing the declarations early in the listing and, in particular, outside and before any of the functions that will make use of them. Once the declarations have been so positioned, the variables by the names declared are known everywhere and can be used by those names anywhere in the listing. While they are needed in some circumstances (see later chapters), global variables are dangerous and must be used sparingly. Once declared, they are known everywhere and must not be used unless their global meaning is intended. Ignoring this rule can result in unpleasant, perhaps even disastrous, interactions among portions of the code that are supposed to be independent.

9.12 Creating and Storing Two-Dimensional Scalar Arrays

We turn now to the storage of one or more arrays representing scalar and vector fields in files with the structure described in Section 9.3.2. Suppose, first, that we wish to explore the two-dimensional
9.12. CREATING AND STORING TWO-DIMENSIONAL SCALAR ARRAYS

Table 9.5: Beginning lines of program irrad.c.

/* PROGRAM irrad.c */
#include <stdio.h> /* Load standard i/o routines */
#include <math.h> /* Load standard math routines */
define narr 1 /* Set number of arrays */
define xdim 50 /* Set dimensions: 50x50x1 */
define ydim 50
#define zdim 1

main()
{
FILE *fptr; /* For file pointer */
float I[xdim][ydim]; /* For irradiance */
float xf, yf; /* For coordinates */
float dx, dy; /* For increments */
float threepi; /* For constant */
int x, y; /* For loop control */

scalar field representing the irradiance produced by Fraunhofer diffraction at a square aperture and
given analytically by the expression

\[ I(x, y) = I_0 \left( \frac{\sin x}{x} \right)^2 \left( \frac{\sin y}{y} \right)^2 \] (9.4)

Here, \( I_0 \) is the irradiance at the center of the pattern \((x = y = 0)\). Let us here decide that we seek
values of \( I(x, y)/I_0 \) over the region \(-3\pi \leq x, y \leq 3\pi\), and that we shall divide each axis into 49
segments of length \( 6\pi/49 \), which will entail evaluating \( I(x, y)/I_0 \) at 50 values of \( x \) and 50 values of
\( y \)—a total of \( 50 \times 50 = 2500 \) values.\(^{23}\)

9.12.2 ... with C

A program in C to create a file containing values of the irradiance given by Eq. (9.4) begins with
a “preamble” in which the program is identified in a comment, necessary standard libraries (input/output and math libraries) are included and necessary variables are declared. The program
continues with a segment in which the internal array is created, and concludes with a segment in
which appropriate labels and the values in that internal array are written into the desired file. We
label the program, include the “include” files, define constants (so the dimensions of the arrays
appear in only one place), and declare variables with the lines listed in Table 9.5.\(^{24}\) Then we set
appropriate values for a few constants with the lines

\(^{23}\)Note that the function to be evaluated is ill-defined at \( x = 0 \) and at \( y = 0 \). We choose 49 rather than 50 segments
so as to avoid having some of the points at which the function is evaluated fall at \( x = 0 \) or \( y = 0 \). Were the function
well defined everywhere, such an awkward number of divisions would not be necessary.

\(^{24}\)In the general structure of Section 9.3.2, we should here be imagining that we are preparing to store one \( 50 \times 50 \times 1 \)
array, and we should thus specify the dimensions as \( I[50][50][1][1] \). Since the last two indices are both 1, however,
they can be omitted.
Table 9.6: Final lines of program irrad.c.

```c
fptr=fopen( "irrad_c.dat", "w" );

fprintf( fptr, "Irradiance; Author: David M. Cook; Date: 15 June 1995\n" );
fprintf( fptr, "Fraunhofer diffraction at square aperture\n" );
fprintf( fptr, "Program described in CPSUP\n" );
fprintf( fptr, "**\n" );
fprintf( fptr, "**\n" );
fprintf( fptr, "%d\n", narr );
fprintf( fptr, "%d\n", xdim );
fprintf( fptr, "%d\n", ydim );
for( y=0; y<ydim; y++ )
{
    for( x=0; x<xdim; x++ )
    {
        fprintf( fptr,"%e\n", I[x][y] );
    }
}
fclose( fptr );
```

```c
threepi = 3.0*3.1415926535; /* Set constant */
dx = 2.0*threepi/(xdim-1.0); /* Set increments */
dy = dx;
```

Next, we evaluate the irradiance given by Eq. (9.4) with the lines

```c
for( y=0; y<ydim; y++ )
{
    yf = y*dy - threepi;
    for( x=0; x<xdim; x++ )
    {
        xf = x*dx - threepi;
        I[x][y] = pow( sin(xf)/xf, 2.0 ) * pow( sin(yf)/yf, 2.0 );
    }
}
```

Finally, we open the file, write the (nine) heading lines and then the irradiance into the file, and close the file with the lines A full listing of this program will be found in Section 9.B. The program itself is stored in the file irrad.c in the directory $HEAD/cc. Once that file has been copied to the default directory, it can be compiled, linked, and run with commands as described in the Local Guide. In UNIX, those commands might be

---

25Program notes: (1) With xdim=50, the loop controlled by the statement for(x=0; x<xdim; x++) is executed first for x=0, then for x=1, x=2, ..., continuing until it has been executed for x=49, which is the index of the xdim\textsuperscript{th} element. (2) With x and y ranging from 0 to 49 and dx and dy defined as 6\pi/49, xf and yf range from -3\pi to 3\pi in 49 equal steps. (3) Note the position of the statement evaluating yf. For computational efficiency, it has been placed in the outer loop. (4) The function pow(r,t) evaluates $r^t$. 

9.13. CREATING AND STORING THREE-DIMENSIONAL SCALAR ARRAYS

cc -o irrad.xc irrad.c -lm
./irrad.xc

Once this (or the appropriate alternative) sequence of commands has been executed by the operating system, the file irrad.c.dat will exist in the default directory.

9.13 Creating and Storing Three-Dimensional Scalar Arrays

As an example of a three-dimensional scalar function, we choose the normalized probability density \( p(x, y, z) \) for the electron in the \((n,l,m) = (3, 2, 0)\) state in the hydrogen atom. This probability density is given as a function of Cartesian coordinates \((x, y, z)\) with the nucleus located at the origin by

\[
p(x, y, z) = \frac{1}{2\pi(27)^3} \rho^4 e^{-2\rho/3} \left( \frac{3z^2}{\rho^2} - 1 \right)^2 = \frac{1}{2\pi(27)^3} e^{-2\rho/3} (9z^4 - 6z^2\rho^2 + \rho^4)
\]

where the coordinates are all measured in units of the Bohr radius and

\[
\rho = \sqrt{x^2 + y^2 + z^2}
\]

This function of three variables is commonly visualized either by focusing on the function in various planes intersecting the three-dimensional volume (thereby reducing the display to a family of two-dimensional displays) or by displaying various contour surfaces. Programs producing such displays need a three-dimensional array of values as input. Suppose, then, we anticipated using some graphical visualization program to explore the quantum probability given by Eq. (9.6). Let us decide to determine values of \( p(x, y, z) \) over the region \(-10 \leq x, y, z \leq 10\), dividing each axis into 29 segments, which will entail evaluating \( p(x, y, z) \) at 30 values of \( x \), 30 values of \( y \), and 30 values of \( z \)—a total of \( 30 \times 30 \times 30 = 27000 \) values.

Since the programs discussed in this section differ very little from those presented in Section 9.12, we elect here to include full code only in the listings in the appendices. We comment in the text only on major differences and subtleties.

9.13.2 ... with C

A C program for creating and storing the necessary three-dimensional scalar array is similar to the program presented in Section 9.12.2. We need one \( 30 \times 30 \times 30 \) array, each element of which is the probability density at the corresponding point in the three-dimensional space. The program, a full listing of which will be found in Section 9.1, is named pdens.c and contains sections

- Identifying the program, including standard libraries, defining constants, and declaring necessary variables. To facilitate translating integers \( x, y, \) and \( z \) into the corresponding real coordinates \( xf, yf, \) and \( zf \), we introduce symbols \([ \text{xrange,yrange,zrange} \) for the interval on each axis and \([ \text{x0,y0,z0} \) for the lowest value of each coordinate. Further, since only one array is to be stored, the array \( P \) for the probability densities is dimensioned at \( P[30][30][30] \) rather than at \( P[30][30][30][1] \). Finally, we introduce variables \( xs, ys, \) and \( zs \) for the squares of the coordinates and \( \rho \) and \( \rho^2 \) for the radial coordinate and its square.

- Evaluating the probability density given by Eq. (9.5). Basically, we need three loops, one ranging over each coordinate from its initial value \([ x0,y0,z0 \) to its final value \([ x0,y0,z0 + xrange, yrange, zrange \) \). In the present case, it is convenient (and computationally efficient) to evaluate each coordinate and then its square as far from the innermost loop as possible and then to evaluate the radial coordinate and its square before evaluating the probability density itself.


- Opening the file, writing the heading lines and the probability densities into the file, and closing the file.

As with previous programs, pdens.c can be copied from the directory $HEAD/cc into your own directory, compiled, linked, and run or—should this be necessary for your application—edited before being compiled, linked, and run to create the desired file. Once the program has been executed, the file pdens.c.dat will exist in the default directory.

9.14 Creating and Storing Two-Dimensional Vector Arrays

Display of a vector field in two dimensions requires a two-dimensional vector array, i.e., a two-dimensional array, each of whose elements is itself a vector. Instead of creating a single two-dimensional vector array, however, we elect to construct a pair of two-dimensional scalar arrays, one for each component of the vector. The first array contains the first component of the vector field at each point on a grid covering the region of interest and the second array contains the second component of the vector field on that same grid.

As an example of a two-dimensional vector field, we choose the magnetic field in a transverse electric (TE) electromagnetic wave propagating in a waveguide with perfectly conducting walls (though we choose also to compute and save the associated electric field). As shown in Fig. 9.4, we take the waveguide to be oriented so that the wave propagates in the positive $x$ direction. Let the guide have a rectangular cross-section with dimensions $(b,d)$, i.e., $(0 < y < b, 0 < z < d)$. Further, let the wave be polarized in the $z$ direction and consider the particular TE waves whose electric field does not depend on $z$. In mksa units, the (complex) fields in this guide are given by

\[ E(r,t) = A \sin \frac{n\pi y}{b} e^{i(\kappa_x x - \omega t)} \mathbf{k} \]  
\[ H(r,t) = \nabla \times E = \frac{A}{i \omega \mu_0} \left( \frac{n\pi y}{b} \cos \frac{n\pi y}{b} \hat{i} - i\kappa_x \sin \frac{n\pi y}{b} \hat{j} \right) e^{i(\kappa_x x - \omega t)} \]  

For instructions on how to compile and run the program, see Section 9.12.2 and the Local Guide.

For a derivation of the magnetic field for a wave propagating in the $z$ direction and a discussion on wave guides in general, see *The Theory of the Electromagnetic Field* by David M. Cook (Prentice-Hall, Englewood Cliffs, NJ, 1975) or *Introduction to Electrodynamics* by David J. Griffiths (Prentice-Hall, Upper Saddle River, NJ, 1999), Third Edition. The first of these books, out of print since the early 1990’s, became available in a Dover reprint in January, 2003, but is now (January 2017) also out of print. It may still be available in your local library.
where \( n = 1, 2, 3, \ldots \) and, with \( c \) standing for the speed of light,
\[
\kappa_x^2 = \frac{\omega^2}{c^2} - \left( \frac{n\pi}{b} \right)^2
\]  
(9.9)

If we focus on the physical fields (real parts of \( E \) and \( H \)) specifically at \( t = 0 \), we find that
\[
\begin{align*}
E(r) & = A \cos \kappa_x x \sin \frac{n\pi y}{b} \hat{k} \\
H(r) & = \frac{An\pi}{\omega \mu_0 b} \sin \kappa_x x \cos \frac{n\pi y}{b} \hat{i} - \frac{A\kappa_x}{\omega \mu_0} \cos \kappa_x x \sin \frac{n\pi y}{b} \hat{j}
\end{align*}
\]  
(9.10)

To simplify these equations further, we divide \( E \) by \( A \) and \( H \) by \( A \frac{n\pi}{\omega \mu_0 b} \), and we elect to measure lengths in units of \( b \) by introducing the variables \( y' = y/b \) and \( x' = x/b \). Equation (9.10) then becomes
\[
\begin{align*}
E(r) & = \frac{E(r)}{A} = \cos \kappa_x b x' \sin n\pi \frac{y'}{b} \hat{k} \\
H(r) & = \frac{H(r)}{A \frac{n\pi}{\omega \mu_0 b}} = \sin \kappa_x b x' \cos n\pi \frac{y'}{b} \hat{i} - \frac{\kappa_x b}{n\pi} \cos \kappa_x b x' \sin n\pi \frac{y'}{b} \hat{j}
\end{align*}
\]  
(9.11)

and we find that the field of interest depends on two parameters \( n \) and \( \kappa_x b \). Further, Eq. (9.9) imposes the constraint
\[
\left( \frac{\kappa_x b}{n\pi} \right)^2 = \left( \frac{\omega b}{n\pi c} \right)^2 - 1
\]  
(9.12)

on these parameters. The only non-zero components of the fields in Eq. (9.11) are
\[
\begin{align*}
\overline{E}_z & = \cos \kappa_x b x' \sin n\pi \frac{y'}{b} \\
\overline{H}_x & = \sin \kappa_x b x' \cos n\pi \frac{y'}{b} \\
\overline{H}_y & = -\frac{\kappa_x b}{n\pi} \cos \kappa_x b x' \sin n\pi \frac{y'}{b}
\end{align*}
\]  
(9.13)

For the sake of a specific example, we choose \( b = 1 \) and \( n = 2 \), and then we choose \( \omega \) so that \( \kappa_x b/2\pi \) turns out to have the value 1. With these choices, we find that the fields we seek to display are given by
\[
\begin{align*}
\overline{E}_z & = \cos 2\pi x' \sin 2\pi \frac{y'}{b} \\
\overline{H}_x & = \sin 2\pi x' \cos 2\pi \frac{y'}{b} \\
\overline{H}_y & = -\cos 2\pi x' \sin 2\pi \frac{y'}{b}
\end{align*}
\]  
(9.14)

where \( x' \) can range over any values—we choose \( 0 \leq x' \leq 1 \)—but, to be inside the guide, \( \overline{y}' \) is confined to the region \( 0 \leq \overline{y}' \leq 1 \). Each component of these fields can now be represented by a two-dimensional array. The \( H \) field, which has two non-zero components, then is translated into two such arrays; the \( E \) field, which has only one non-zero component, requires only one such array. To be explicit, we determine values of \( \overline{H}_x(x, \overline{y}), \overline{H}_y(x, \overline{y}), \) and \( \overline{E}_z(x, \overline{y}) \) over the region \( 0 \leq x, \overline{y} \leq 1 \), dividing each axis into 30 segments of length 1.0/29.0, which will entail evaluating the field components at 30 values of \( x \) and 30 values of \( \overline{y} \)—a total of \( 30 \times 30 = 900 \) values. Normally, these field components would be stored in three \( 30 \times 30 \) two-dimensional arrays. Because of our declared file format, however, we must view ourselves as needing storage for three \( 30 \times 30 \times 3 \) three-dimensional arrays. We shall, however, view the structure as a single \( 30 \times 30 \times 1 \times 3 \) array \( H \), with \( H(*, *, 1, 1) \) storing \( \overline{H}_x \), \( H(*, *, 1, 2) \) storing \( \overline{H}_y \), and \( H(*, *, 1, 3) \) storing \( \overline{E}_z \).\footnotemark

\footnotetext{We have supposed indices starting at 1. In languages where indices start at 0, the associations would, of course, be \( \overline{H}(*, *, 0, 0) \) storing \( \overline{H}_x \), \( \overline{H}(*, *, 0, 1) \) storing \( \overline{H}_y \), and \( \overline{H}(*, *, 0, 2) \) storing \( \overline{E}_z \).}
Table 9.7: Opening lines of program wavegd.c.

/* PROGRAM wavegd.c */
#include <stdio.h> /* Load standard i/o routines */
#include <math.h> /* Load standard math routines */
#define narr 3 /* Set number of arrays */
#define xdim 30 /* Set dimensions: 30x30x1 */
#define ydim 30
#define zdim 1

main()
{
FILE *fptr; /* For file pointer */
float H[xdim][ydim][zdim][narr]; /* For H */
float xf, yf; /* For coordinates */
float sxf, cxf, syf, cyf; /* For sin, cos */
float dx, dy; /* For increments */
float twopi; /* For constant */
int n, x, y; /* For loop control */

9.14.2 ... with C

A C program for creating and storing the necessary arrays is similar to the program discussed in Section 9.12.2. After identifying the program in a comment, including necessary standard “include” libraries (input/output and math libraries), defining assorted constants, and declaring necessary variables with the lines listed in Table 9.7, the main routine continues by setting appropriate values for a few other quantities with the lines

twopi = 2.0*3.1415926535; /* Set constant */
dx = twopi/(xdim-1.0) /* Set increments */
dy = dx;

Then we evaluate the field components in Eq. (9.14) with the lines listed in Table 9.8.\textsuperscript{29,30,31} Finally, we open the file, write the (nine) heading lines and then the array into the file, and close the file with the lines listed in Table 9.9. A full listing of this program will be found in Section 9.1L The program itself is stored in the file wavegd.c in the directory $\texttt{HEAD/cc}$. It can be copied into your own directory, compiled, linked, and run or—should this be necessary for your application—edited before being compiled, linked, and run to create the desired file.\textsuperscript{32} Once the program has been executed, the file wavegd.c.dat will exist in the default directory.

9.15 Creating and Storing Three-Dimensional Vector Arrays

Display of a vector field in three dimensions requires a three-dimensional vector array, i.e., a three-dimensional array, each of whose elements is itself a vector. Instead of creating a single three-dimensional vector array, however, we elect to construct a triplet of three-dimensional scalar arrays,\textsuperscript{32} For instructions on how to compile and run the program, see Section 9.12.2 and the Local Guide.
Table 9.8: Coding to calculate the field in program `wavegd.c`.

```c
for( x=0; x<xdim; x++ )
{
    xf = x*dx;
sxf = sin(xf);
cxf = cos(xf);
    for( y=0; y<ydim; y++ )
    {
        yf = y*dy;
syf = sin(yf);
cyh = cos(yh);
        H[x][y][0][0] = sxf * cyh; /* Hx */
        H[x][y][0][1] = -cxf * syf; /* Hy */
        H[x][y][0][2] = -H[x][y][0][1]; /* Ez */
    }
}
```

Table 9.9: Closing lines of program `wavegd.c`.

```c
fptr = fopen( "wavegd_c.dat", "w" );
fprintf( fptr, "Waveguide; Author: David M. Cook; Date: 15 June 1995\n" );
fprintf( fptr, "H and E fields in rectangular waveguide\n" );
fprintf( fptr, "Program described in CPSUP\n" );
fprintf( fptr, "**\n" );
fprintf( fptr, "**\n" );
fprintf( fptr, "%d\n", narr );
fprintf( fptr, "%d\n", xdim );
fprintf( fptr, "%d\n", ydim );
fprintf( fptr, "%d\n", zdim );

for( n=0; n<narr; n++ )
{
    for( y=0; y<ydim; y++ )
    {
        for( x=0; x<xdim; x++ )
        {
            fprintf( fptr, "%e\n", H[x][y][0][n] );
        }
    }
}
fclose( fptr );
```
one for each component of the vector. The first array contains the first component of the vector field at each point on a grid covering the region of interest, the second array contains the second component of the vector field on that same grid, and the third array contains the third component of the vector field. This section describes convenient ways to produce such triplets of arrays and to write them into ASCII files for transfer to other programs.

As an example of a three-dimensional vector field, we choose the electric field produced by a quadrupole consisting of four charges at the corners of a square of side 2\(a\) with its center at the origin and its plane in the \(xy\) plane. Choosing to measure the coordinates \(x, y\) and \(z\) in units of \(a\), we find that this field is given by

\[
\left(\frac{E_x(x, y, z), E_y(x, y, z), E_z(x, y, z)}{q/4\pi\epsilon_0a^2}\right) = \left[\frac{(x-1,y-1,z)}{[(x-1)^2+(y-1)^2+z^2]^{3/2}} - \frac{(x+1,y-1,z)}{[(x+1)^2+(y-1)^2+z^2]^{3/2}} + \frac{(x+1,y+1,z)}{[(x+1)^2+(y+1)^2+z^2]^{3/2}} - \frac{(x-1,y+1,z)}{[(x-1)^2+(y+1)^2+z^2]^{3/2}}\right]
\]  

(9.15)

To be explicit, we determine values of the field components over the region \(-2.0 \leq x, y, z \leq 2.0\), dividing each axis into 29 segments, which will entail evaluating the components at 30 values of \(x\), 30 values of \(y\), and 30 values of \(z\)—a total of 30 \(\times\) 30 \(\times\) 30 = 27000 values. Because of our declared file format, however, we must view ourselves as needing storage for three 30 \(\times\) 30 \(\times\) 30 three-dimensional arrays. We shall, however, view the structure as a single 30 \(\times\) 30 \(\times\) 30 array \(E\), with \(E(*,*,*,1)\) storing \(E_x\), \(E(*,*,*,2)\) storing \(E_y\), and \(E(*,*,*,3)\) storing \(E_z\).

Since the programs discussed in this section differ very little from those presented in Section 9.14, we elect here to include full code only in the listings in the appendices. We comment in the text only on major differences and subtleties.

### 9.15.2 ... with C

A C program for creating and storing three-dimensional vector arrays is similar to the other C programs presented so far, in particular the three-dimensional scalar program in Section 9.13.2. This time, however, we must exploit the fourth dimension that our file format allows us. Each point in \(xyz\) space has three components associated with it, so we set the fourth dimension \(n\) (the number of arrays) equal to 3 to allow for those components. The program, a full listing of which will be found in Section 9.O, is named \texttt{quadpole.c} and contains sections

- Identifying the program, including standard libraries, defining constants, and declaring necessary variables. To facilitate translating integers \(x\), \(y\), and \(z\) into the corresponding real coordinates \(xf, yf\), and \(zf\), we introduce symbols \([\texttt{xrange}, \texttt{yrange}, \texttt{zrange}]\) for the interval on each axis and \([\texttt{x0}, \texttt{y0}, \texttt{z0}]\) for the lowest value of each coordinate. Since three arrays are to be stored, the array \(E\) for the electric field components is dimensioned at \(E[30][30][30]\), with \(E_x\) stored in \(E[*][*][0]\), \(E_y\) stored in \(E[*][*][1]\), and \(E_z\) stored in \(E[*][*][2]\). Finally, we introduce variables \(xfm, xfp, yfm\) and \(yfp\) for \(x-1\), \(x+1\), \(y-1\), and \(y+1\), respectively; variables \(xms, xps, yms, yps\), and \(zms, zps\) for \((x-1)^2\), \((x+1)^2\), \((y-1)^2\), \((y+1)^2\), and \(z^2\), respectively; and variables \(rmm, rpm, rp\) and \(rpp\) for the four denominators in Eq. 9.15.

- Evaluating the electric field components given by Eq. (9.15). Basically, we need three loops, one ranging over each coordinate from its initial value \([\texttt{x0}, \texttt{y0}, \texttt{z0}]\) to its final value \([\texttt{x0}, \texttt{y0}, \texttt{z0}] + [\texttt{xrange}, \texttt{yrange}, \texttt{zrange}]\). In the present case, it is convenient (and computationally efficient) to evaluate as many quantities as possible as far from the center of the loop as possible and then to evaluate the denominators before evaluating the field components themselves.

\[\text{We have supposed indices starting at 1. In languages where indices start at 0, the associations would, of course, be } E(*,*,0,0) \text{ storing } E_x, E(*,*,1,0) \text{ storing } E_y, \text{ and } E(*,*,2,0) \text{ storing } E_z.\]
• Opening the file, writing the heading lines and the field components into the file, and closing the file.

As with previous programs, quadpole.c can be copied from the directory \$HEAD/cc into your own directory, compiled, linked, and run or—should this be necessary for your application—edited before being compiled, linked, and run to create the desired file.\(^{34}\) Once the program has been executed, the file quadpole_c.dat will exist in the default directory.

9.16 Reading Files

Full use of the files that we have learned how to create in the previous sections, of course, depends on an ability to read files created with a program in one language into a program in another—or possibly the same—language. This section will lay out the necessary features.

9.16.2 ... with C

Data can be read from the files in C with a variety of statements. Unfortunately, in the simplest situations, arrays must be dimensioned at the time of compilation. Thus, for the simplest programs, we must know the size of the array(s) to be read when we write the program.\(^{35}\) In this subsection, then, we will develop a C program to read the file irrad_c.dat created in Section 9.12.2, understanding that programs to read other files will be similar.

The task involves only a few steps. First, we must declare a suitable pointer to be used in referring to the file and open the file for read access with the statements

```c
FILE *fptr;
fptr = fopen( "irrad_c.dat", "r" );
```

Then we must declare appropriate variables and read past the five lines of comments with statements like

```c
char ln1[120], ln2[120], ln3[120], ln4[120], ln5[120];
int narr, ix, iy, iz;
fgets( ln1, 120, fptr );
fgets( ln2, 120, fptr );
fgets( ln3, 120, fptr );
fgets( ln4, 120, fptr );
fgets( ln5, 120, fptr );
```

Next, we declare appropriate variables and read the four dimensions from the file with the statements

```c
int narr, ix, iy, iz;/fsfscanf( fptr, "%d %d %d %d", narr, ix, iy, iz );
```

(We will not—though for security we probably should—verify that the values read from the file for these integers actually are the values—1, 50, 50, 1—we have used for the dimensions of the array.) Finally, we declare an appropriate array for the data, declare integers for the loop indices, read the values from the file, and close the file with the statements

\(^{34}\) For instructions on how to compile and run the program, see Section 9.12.2 and the Local Guide.

\(^{35}\) Reading the dimensions of the necessary array from the file at execution time is simple enough but creating the array at that time is much trickier. We choose not to address the necessary strategy in this book.
Table 9.10: Algorithm for Exercise 9.1.

**PROGRAM PARK**
Obtain date and time of entry to lot
Obtain date and time of exit from lot
Determine time in lot in days, hours, and minutes
Report time in lot
IF time in lot less than one day
    THEN calculate fee for less than one day
    ELSE calculate fee for one day or longer
END_IF
Report fee
END_PROGRAM

```c
int n, i, j, k;
float I[50][50][1][1];
for( n=0; n<narr; n++ )
    for( k=0; k<iz; k++ )
        for( j=0; j<iy; j++ )
            for( i=0; i<ix; i++ )
                fscanf( fptr, "%g", &I[i][j][k][n] );
fclose( fptr );
```

Execution of these statements will recreate the variable I as it was when the file *irrad.c.dat* was created in Section 9.12.2. A fully commented and properly ordered listing of the program *read_irrad.c* is presented in Section 9.U.

**9.17 References**


**9.18 Exercises**

9.1. Pricing at an airport parking lot is as follows: 50 cents for the first half hour, 35 cents for the second half hour, and 25 cents for each subsequent hour (or fraction thereof) to a maximum of 250 cents per 24-hour period. The higher charge for the first hour applies *only* on the first day. Make the algorithm listed in Table 9.10 for determining the parking fee for a particular patron more explicit by expanding the two statements in the *IF-THEN-ELSE* structure. *Hint*: Work through several numeric examples by hand, noting particularly all decisions that you must make in order to know what arithmetic to do. *Optional*: Describe a procedure for determining the time in the lot from the dates and times of entry and exit. Assume first that the two dates are in the same month, but then give some thought to generalizing your procedure to handle cases where the two dates span two or more months or years.

9.2. Figure 9.5 shows three different alternative structures. Express each structure using (a) *only CASE* structures and (b) *only IF-THEN-ELSE* structures. In these figures, T, F, C, and B stand for true,
9.18. EXERCISES

Figure 9.5: Figure for Exercise 9.2.

false, condition, and block of statements, respectively. Use proper indentation as illustrated in the examples.

9.3. Identify the basic actions performed by the automatic pin setting apparatus at the end of a bowling alley. Then write an algorithm to control the operation of this device.

9.4. For the game of bowling, identify appropriate elementary action statements and then write an algorithm that will accept the number of pins knocked over with each ball and report the frame-by-frame scores.

9.5. (a) Cast Algorithm (6) of Section 9.2 in a form more specific to finding the largest integer in a list of integers.
   (b) The algorithm shown in Table 9.11 is an alternative to the algorithm deduced in (a). Essentially, an alternative method of initializing $\text{LARGEST}$ is adopted. Write a few sentences identifying the advantages and disadvantages of the two different methods.

9.6. Basing your work on Algorithm (6) of Section 9.2, write an algorithm that will obtain words one at a time and ultimately report the word that would occur first if the list were to be alphabetized.

9.7. Basing your work on Algorithm (6) of Section 9.2, write an algorithm that will obtain words one at a time and ultimately report (1) the word that would appear last if the list were alphabetized,
### Table 9.11: Algorithm for Exercise 9.5.

```
SENTINEL% ← ⟨agreed-upon special value⟩
LARGEST% ← ⟨assumed fictitious integer known to occur before any possible real integer in the list⟩
LOOP
  READ ITEM%
  EXIT_LOOP WHEN ITEM% = SENTINEL%
  IF ITEM% > LARGEST%
    THEN LARGEST% ← ITEM%
  END_IF
END_LOOP
Report LARGEST%
```

(2) the word that would appear first if the list were alphabetized, (3) the total number of words given, and (4) the position of each extreme word in the original list. Only one pass through the list is permitted.

**9.8.** Write an algorithm that will find and report all triplets of positive integers (zero excluded) \(A\%, B\%, C\%\) satisfying \(A\% \cdot A\% + B\% \cdot B\% = C\% \cdot C\%\), subject to the restriction that \(A\%, B\%,\) and \(C\%\) shall all be smaller than some value \(\text{MAXNUM}\%\) supplied as input. *Hint*: Systematically examine all possibilities, but do so thoughtfully. For example, there is no point in examining cases for which \(C\% \leq A\%\) or \(C\% \leq B\%.\) Express your loops so that these cases (and any others that you can reject *a priori*) will not even be considered. *Optional*: For \(\text{MAXNUM}\% = 20\%,\) determine the number of executions of your innermost loop.

**9.9.** Suppose you have \(N\%\) cards laid out in a row on a table. On each card is a single word. Determine the end result of applying the mystery procedure laid out in Table 9.12 to that array of cards and choose a suitable name for the procedure.

**9.10.** Let the digits in an integer be counted from the left end of the integer, i.e., in the four-digit number “4358”, call “4” digit 1, “3” digit 2, “5” digit 3, and “8” digit 4. Determine the function of the mystery procedure in Table 9.13 and choose a suitable name for the procedure.

**9.11.** Starting with the approximation

\[
\frac{du}{dx} \Big|_{x + \frac{1}{2}\Delta x} \approx \frac{u(x + \Delta x) - u(x)}{\Delta x}
\]

deduce the finite difference approximation

\[
\frac{d^2u}{dx^2} \bigg|_x \approx \frac{u(x + \Delta x) - 2u(x) + u(x - \Delta x)}{\Delta x^2}
\]

for the second derivative.

**9.12.** Write, compile, and test a program that asks for the input of a temperature in Celsius and prints out the corresponding temperature in Fahrenheit. To make it a bit more of a challenge, write the program in such a way that it asks repeatedly for Celsius temperatures until the temperature 9999 is entered, at which point the program terminates smoothly.

**9.14.** (a) Copy laplace_file.c from $HEAD/cc to your directory, naming it laplace15_file.c. Then compile, link, and run laplace15_file.f to generate the file laplace.c.dat, which you should rename laplace15.c.dat.

(b) Produce the program laplace29_file.f by copying laplace15_file.c to the new file and editing the new file so that the program, when run, generates a solution on a 29 × 29 grid.
9.18. EXERCISES


<table>
<thead>
<tr>
<th>Procedure Procedure</th>
<th>SCANEND% ← N%</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOOP CARD% ← 1%</td>
<td></td>
</tr>
<tr>
<td>Obtain word on card CARD% and store in WORD$</td>
<td></td>
</tr>
<tr>
<td>LATEST_WORD$ ← WORD$</td>
<td></td>
</tr>
<tr>
<td>LATEST_CARD% ← CARD%</td>
<td></td>
</tr>
<tr>
<td>LOOP CARD% ← CARD% + 1%</td>
<td></td>
</tr>
<tr>
<td>Obtain word on card CARD% and store in WORD$</td>
<td></td>
</tr>
<tr>
<td>IF WORD$ occurs after LATEST_WORD$ THEN BEGIN_BLOCK</td>
<td></td>
</tr>
<tr>
<td>LATEST_WORD$ ← WORD$</td>
<td></td>
</tr>
<tr>
<td>LATEST_CARD% ← CARD%</td>
<td></td>
</tr>
<tr>
<td>END_BLOCK END_IF EXIT_LOOP WHEN CARD% = SCANEND%</td>
<td></td>
</tr>
<tr>
<td>END_LOOP Exchange card LATEST_CARD% with card SCANEND%</td>
<td></td>
</tr>
<tr>
<td>SCANEND% ← SCANEND% - 1% EXIT_LOOP WHEN SCANEND% = 1%</td>
<td></td>
</tr>
<tr>
<td>END_LOOP END_PROCEDURE</td>
<td></td>
</tr>
</tbody>
</table>

(c) Compile, link, and run laplace29_file.f and rename the output file to laplace30.f.dat.

(d) Import both laplace15_c.dat and laplace30_c.dat into an available program for graphical visualization and then

i. generate a graphical display of each solution, either a contour map in the $xy$ plane or a surface plot over the $xy$ plane (or perhaps both). In either case (or both cases), make sure the axes are labeled correctly with the proper coordinate values. Warning: This latter requirement is a bit subtle. Beware.

ii. develop a way to compare the two solutions at those grid points that are common to the two and display the differences graphically.

(e) Copy the file laplace29_file.c to a new name—your choice—and then edit that file so that the program it conveys monitors the change from one iteration to the next and displays on the screen the maximum absolute value of the change that occurs during the course of each iteration.

(f) Edit the last program again so that iteration is stopped when the maximum change falls below a tolerance that is specified as input when the program is run (or—to prevent infinite loops—when the number of iterations exceeds some maximum value). Arrange for the program to display the number of iterations carried out when the solution finally converges. Compile, link, and run this last program and explore the way the number of iterations varies with the tolerance specified. Was 300 iterations as a trial in the original programs vast overkill?

9.15. Consider two circular current loops, each of radius $a$ and lying with its center on and its plane perpendicular to the $z$ axis. The first loop is centered at the point $(0, 0, b)$ and the second loop is centered at the point $(0, 0, -b)$. The axial component of the magnetic field at the point $(0, 0, z)$ is given by the equation

$$B(z) = \frac{1}{2} B_0 \left( a^2 + b^2 \right)^{3/2} \left[ \frac{1}{a^2 + (z + b)^2}^{3/2} + \frac{1}{a^2 + (z - b)^2}^{3/2} \right]$$

```plaintext
PROCEDURE ??????
Obtain a positive integer A% from a friend
Obtain a (second) positive integer B% from a friend
CASE
  OF A% has more digits than B% DO
    LOOP
      Add digit 0 in front of B% and call result B%
      EXIT_LOOP WHEN B% and A% have same number of digits
    END_LOOP
  OF B% has more digits than A% DO
    LOOP
      Add digit 0 in front of A% and call result A%
      EXIT_LOOP WHEN A% and B% have same number of digits
    END_LOOP
END_CASE
WRITE A% on a piece of paper
WRITE B% under A% with corresponding digits in same column
Draw a line under B%
DIGIT% ← number of digits in either number
CARRY% ← 0%
LOOP
  SUM% ← digit DIGIT% of A% + digit DIGIT% of B% + CARRY%
  IF SUM% < 10%
    THEN CARRY% ← 0%
  ELSE BEGIN_BLOCK
    CARRY% ← 1%
    SUM% ← SUM% - 10%
  END_BLOCK
  END_IF
WRITE SUM% under digit DIGIT% of B%
DIGIT% ← DIGIT% - 1%
EXIT_LOOP WHEN DIGIT% = 0%
END_LOOP
IF CARRY% = 1%
  THEN WRITE CARRY% in front of all digits beneath line
END_IF
END_PROCEDURE
```

where \( B_0 \) is the magnetic field at the origin. This field can be considered as a function not only of \( z \), the coordinate of a point on the \( z \) axis, but also of \( b \), (half) the separation of the two loops. Create a file conforming to the structure described in Section 9.3.2 and containing values of this function seen as a two-dimensional scalar function of \( z/a \) and \( b/a \). *Suggestion:* Write values of \( B/B_0 \) into the file.

9.16. The trajectory of a particle in three-dimensional space is given parametrically as a function of time \( t \) by the position vector

\[
r = x(t) \hat{i} + y(t) \hat{j} + z(t) \hat{k}
\]
You desire to fathom out the general character of this trajectory by using a graphical visualization tool that does not have much computational capability. Thus, you must generate the data using one tool but will visualize the trajectory with another tool. You elect to use an ASCII file to communicate the data from the first tool to the second. Suppose that the ASCII file produced by the first tool is to be structured as follows:

- five lines of text describing the contents of the file and its origin,
- one line containing the number of points \(N\) on the trajectory included in the file, and
- \(N\) lines, each of which contains four floating point values separated by commas, those values being in order \(t, x(t), y(t),\) and \(z(t)\) for a point on the trajectory. (The \(N\) lines are ordered by increasing value of \(t.\))

Describe a general procedure to create this file and then implement that procedure in at least one language of your choice, testing your program(s) with the trajectory given by

\[
\mathbf{r} = \cos t \hat{i} + \sin t \hat{j} + 0.1t \hat{k}
\]

which describes the path followed by a charged particle in a constant magnetic field along the \(z\) axis.

9.17. Following the pattern illustrated in Section 9.13, use at least one language to create a file containing values for at least one of the three-dimensional scalar fields

\[
p_{3,1,0}(x, y, z) = \frac{8}{(27)^{2/3}} \rho^2 \left(1 - \frac{\rho^2}{6}\right)^2 e^{-2\rho^3/3} \cos^2 \theta
\]

\[
p_{3,1,1}(x, y, z) = \frac{4}{(27)^{2/3}} \rho^2 \left(1 - \frac{\rho^2}{6}\right)^2 e^{-2\rho^3/3} (1 - \cos^2 \theta)
\]

\[
p_{3,2,1}(x, y, z) = \frac{3}{(27)^{2/3}} \rho^4 e^{-2\rho^3/3} \cos^2 \theta (1 - \cos^2 \theta)
\]

\[
p_{3,2,2}(x, y, z) = \frac{3}{4(27)^{2/3}} \rho^4 e^{-2\rho^3/3} (1 - \cos^2 \theta)^2
\]
giving the probability density for the hydrogen states \((n, l, m) = (3, 1, 0), (n, l, m) = (3, 1, 1), (n, l, m) = (3, 2, 1),\) and \((n, l, m) = (3, 2, 2)\). These fields are expressed in dimensionless form, where \(\rho\) is the radial coordinate in units of the Bohr radius. In terms of the Cartesian coordinates \(x, y, z, \rho = \sqrt{x^2 + y^2 + z^2}\) and \(\cos \theta = z/\rho\). Hint: To avoid divisions by zero, recast the expressions in terms of \((x, y, z)\) explicitly before evaluating any of them.

9.18. The (gauge) pressure \(p(x, y, z, t)\) inside a cubical box located in the region \(0 \leq x, y, z \leq a\) is given by

\[
p(x, y, z, t) = A \sin \left(\frac{l\pi x}{a}\right) \sin \left(\frac{m\pi y}{a}\right) \sin \left(\frac{n\pi z}{a}\right) \cos \omega_{lmn} t
\]

where \(l, m,\) and \(n\) are positive integers. Using at least one language, create files containing the pressure distribution inside the box at \(t = 0\) for several different values of \(l, m,\) and \(n.\)

9.19. A point charge of strength \(q\) is located on the \(y\) axis at \(\mathbf{r}_+ = a \hat{j}\) and a point charge of strength \(-q\) is located on the \(y\) axis at \(\mathbf{r}_- = -a \hat{j}\). The electric field \(\mathbf{E}\) at the point \(\mathbf{r} = x \hat{i} + y \hat{j}\) in the \(xy\) plane is given in mks units by

\[
\mathbf{E}(x, y) = \frac{q}{4\pi \varepsilon_0} \left[\frac{x \hat{i} + (y - a) \hat{j}}{[x^2 + (y - a)^2]^{3/2}} - \frac{x \hat{i} + (y + a) \hat{j}}{[x^2 + (y + a)^2]^{3/2}}\right]
\]

Expressing coordinates in terms of the dimensionless variables \(\bar{x} = x/a\) and \(\bar{y} = y/a\) and measuring \(\mathbf{E}\) in the unit \(q/(4\pi \varepsilon_0 a^2)\), create a file conforming to the structure described in Section 9.3.2 and containing the \(x\) and \(y\) components of this field over the interval \(-2a \leq x, y \leq 2a\). Divide the interval in each coordinate direction into about 25 segments, but choose the precise number carefully so as to avoid division by zero in evaluating the field at any point. Optional: Read your file into a suitable graphical display program and produce graphs showing the character of this field.
9.20. The velocity field of an object rotating about the $z$ axis with angular momentum $\omega$ is given in terms of the angular velocity and the position vector $\mathbf{r}$ by the expression

$$\mathbf{v} = \omega \times \mathbf{r} = \omega \mathbf{k} \times (x \mathbf{i} + y \mathbf{j} + z \mathbf{k}) = \omega (-y \mathbf{i} + x \mathbf{j})$$

Choosing a unit of length $a$ and expressing coordinates in units of $a$ and velocities in units of $\omega a$, create a file conforming to the structure described in Section 9.3.2 and containing the $x$ and $y$ components of this field over the interval $-2a \leq x, y \leq 2a$. Choose the number of divisions in each coordinate direction so as to generate a display which is neither too sparse to be useful nor too busy to be intelligible. Optional: Read your file into a suitable graphical display program and produce graphs showing the character of this field.

9.21. A point charge of strength $q$ is located on the $z$ axis at $\mathbf{r}_+ = a \mathbf{k}$ and a point charge of strength $-q$ is located on the $z$ axis at $\mathbf{r}_- = -a \mathbf{k}$. The electric field $\mathbf{E}$ at the point $\mathbf{r} = x \mathbf{i} + y \mathbf{j} + z \mathbf{k}$ is given in mks units by

$$\mathbf{E}(x, y, z) = \frac{q}{4\pi\epsilon_0} \left[ \frac{x \mathbf{i} + y \mathbf{j} + (z - a) \mathbf{k}}{[x^2 + y^2 + (z - a)^2]^{3/2}} - \frac{x \mathbf{i} + y \mathbf{j} + (z + a) \mathbf{k}}{[x^2 + y^2 + (z + a)^2]^{3/2}} \right]$$

Expressing coordinates in terms of the dimensionless variables $\tilde{x} = x/a$, $\tilde{y} = y/a$, and $\tilde{z} = z/a$ and measuring $\mathbf{E}$ in the unit $q/(4\pi\epsilon_0 a^2)$, create a file conforming to the structure described in Section 9.3.2 and containing the $x$, $y$, and $z$ components of this field over the interval $-2a \leq x, y, z \leq 2a$. Divide the interval in each coordinate direction into about 25 segments, but choose the precise number carefully so as to avoid division by zero in evaluating the field at any point. Optional: Read your file into a suitable graphical display program and produce graphs showing the character of this field.

9.22. Write and test a program to ask for the latitude and longitude of both a point of departure $D$ and a point of arrival $A$ on the surface of the earth and then calculate and print out the “crow-flies” distance along a great circle route from $D$ to $A$. Make sure your program prints the shorter of the two distances, regardless of the location of the points, and make sure your program doesn’t run into difficulties if the two points happen to be at opposite ends of a diameter. Take the earth to be a perfect sphere with a circumference of 24900 miles (radius 3963 miles). For purposes of testing, note that Albany, NY, is at $[43^\circ40' \text{ N}, 73^\circ45' \text{ W}]$; Grand Junction, CO, is at $[39^\circ5' \text{ N}, 108^\circ33' \text{ W}]$; Los Angeles, CA, is at $[34^\circ3' \text{ N}, 118^\circ15' \text{ W}]$; Appleton, WI, is at $[44^\circ16' \text{ N}, 88^\circ25' \text{ W}]$; Calcutta, India, is at $[22^\circ32' \text{ N}, 88^\circ20' \text{ E}]$; Sydney, Australia, is at $[33^\circ52' \text{ S}, 151^\circ12' \text{ E}]$; Paris, France, is at $[48^\circ49' \text{ N}, 2^\circ29' \text{ E}]$; and Stockholm, Sweden, is at $[59^\circ21' \text{ N}, 18^\circ4' \text{ E}]$. 
9.B  Listing of irrad.c

/* PROGRAM irrad.c */

/* The details of this program are described in CPL-620. */

#include <stdio.h> /* Load standard i/o routines */
#include <math.h> /* Load standard math routines */
#define narr 1 /* Set number of arrays */
#define xdim 50 /* Set dimensions: 50x50x1 */
#define ydim 50
#define zdim 1

main()
{

    FILE *fptr; /* For file pointer */
    float I[xdim][ydim]; /* For irradiance */
    float xf, yf; /* For coordinates */
    float dx, dy; /* For increments */
    float threepi; /* For constant */
    int x, y; /* For loop control */

    threepi = 3.0*3.1415926535; /* Set constant */
    dx = 2.0*threepi/(xdim-1.0); /* Set increments */
    dy = dx;

    /***** EVALUATE IRRADIANCE *****/

    for( y=0; y<ydim; y++ )
    {
        yf = y*dy - threepi;
        for( x=0; x<xdim; x++ )
        {
            xf = x*dx - threepi;
            I[x][y] = pow( sin(xf)/xf, 2.0 ) * pow( sin(yf)/yf, 2.0 );
        }
    }

    /***** OPEN FILE IN WRITE MODE; GET POINTER *****/

    fptr = fopen( "irrad_c.dat", "w" );

    /***** WRITE HEADING LINES TO FILE *****/

    printf( fptr, "Irradiance; Author: David M. Cook; Date: 15 June 1995\n" );
    printf( fptr, "Fraunhofer diffraction at square aperture\n" );
    printf( fptr, "Program described in CPSUP\n" );
    printf( fptr, "\n" );
    printf( fptr, "%d
", narr );
    printf( fptr, "%d
", xdim );
fwrite( fptr, "%d\n", ydim );
fwrite( fptr, "%d\n", zdim );

/***** WRITE IRRADIANCE TO FILE *****/

for( y=0; y<ydim; y++ )
{
    for( x=0; x<xdim; x++ )
    {
        fprintf( fptr, "%e\n", I[x][y] );
    }
}

/***** CLOSE FILE *****/

fclose( fptr );
}
/* PROGRAM pdens.c */

/* The details of this program are described in CPL-620. */

#include <stdio.h> /* Load standard i/o routines */
#include <math.h> /* Load standard math routines */
#define narr 1 /* Set number of arrays */
#define xdim 30 /* Set dimensions: 30x30x30 */
#define ydim 30
#define zdim 30
#define twopi 6.28321853 /* Set 2*pi */
#define xrange 20.0 /* Set length of interval in x */
#define yrange 20.0 /* Set length of interval in y */
#define zrange 20.0 /* Set length of interval in z */
#define x0 -10.0 /* Set starting value of x */
#define y0 -10.0 /* Set starting value of y */
#define z0 -10.0 /* Set starting value of z */

main()
{

    FILE *fptr; /* For file pointer */
    float P[xdim][ydim][zdim]; /* For probability density */
    float xf, yf, zf; /* For coordinates */
    float dx, dy, dz; /* For increments */
    float xs, ys, zs; /* For x^2, y^2, z^2 */
    float rho, rhos; /* For radial coordinate */
    float pfactor; /* For constant */
    int x, y, z; /* For loop control */

    pfactor = 1.0/(twopi*27.0*27.0*27.0); /* Set constant */

    dx = xrange/(xdim - 1.0); /* Set increments */
    dy = yrange/(ydim - 1.0);
    dz = zrange/(zdim - 1.0);

    //***** EVALUATE PROBABILITY DENSITY *****/*

    for( z=0; z<zdim; z++ )
    {
        zf = z * dz + z0;
        zs = zf*zf;
        for( y=0; y<ydim; y++ )
        {
            yf = y * dy + y0;
            ys = yf*yf;
            for( x=0; x<xdim; x++ )
            {
                xf = x * dx + x0;
                xs = xf*xf;
                PF; /* Probability density function */
            }
        }
    }

}
rhos = xs + ys + zs;
rho = sqrt(rhos);
P[x][y][z] = pfactor * exp( -2.0*rho/3.0 ) *
              ( 9*zs*zs - 6*zs*rhos + rhos*rhos );
}
}
}

/***** OPEN FILE IN WRITE MODE; GET POINTER *****/

fptr = fopen( "pdens_c.dat", "w" );

/***** WRITE HEADING LINES TO FILE *****/

fprintf( fptr, "Probability density; Author: Chris Schmidt;" );
fprintf( fptr, " Date: 26 June 1995\n" );
fprintf( fptr, "The probability density for the electron in the\n" );
fprintf( fptr, "(n,l,m) = (3,2,0) state in the hydrogen atom.\n" );
fprintf( fptr, "Program described in CPSUP\n" );
fprintf( fptr, "**\n" );

fprintf( fptr, "%d\n", narr );
fprintf( fptr, "%d\n", zdim );
fprintf( fptr, "%d\n", ydim );
fprintf( fptr, "%d\n", xdim );

/***** WRITE PROBABILITY DENSITY TO FILE *****/

for( z=0; z<zdim; z++ )
{
    for( y=0; y<ydim; y++ )
    {
        for( x=0; x<xdim; x++ )
        {
            fprintf( fptr, "%e\n", P[x][y][z] );
        }
    }
}

/***** CLOSE FILE *****/

fclose( fptr );
/* PROGRAM wavegd.c */

/* The details of this program are described in CPL-620. */

#include <stdio.h> /* Load standard i/o routines */
#include <math.h> /* Load standard math routines */
#define narr 3 /* Set number of arrays */
#define xdim 30 /* Set dimensions: 30x30x1 */
#define ydim 30
#define zdim 1
main()
{

/***** DECLARE VARIABLES; EVALUATE CONSTANTS *****/

FILE *fptr; /* For file pointer */
float H[xdim][ydim][zdim][narr]; /* For H */
float xf, yf; /* For coordinates */
float sxf, cxf, syf, cyf; /* For sin, cos */
float dx, dy; /* For increments */
int n, x, y; /* For loop control */
int twopi; /* For constant */
twopi = 2.0*3.1415926535; /* Set constant */
dx = twopi/(xdim-1.0); /* Set increments */
dy = dx;

/***** EVALUATE FIELD COMPONENTS *****/

for( x=0; x<xdim; x++ )
{
    xf = x*dx;
    sxf = sin(xf);
    cxf = cos(xf);
    for( y=0; y<ydim; y++ )
    {
        yf = y*dy;
        syf = sin(yf);
        cyf = cos(yf);
        H[x][y][0][0] = sxf * cyf; /* Hx */
        H[x][y][0][1] = -cxf * syf; /* Hy */
        H[x][y][0][2] = -H[x][y][0][1]; /* Ez */
    }
}

/***** OPEN FILE IN WRITE MODE; GET POINTER *****/

fptr = fopen( "wavegd_c.dat", "w" );

/***** WRITE HEADING LINES TO FILE *****/
// WRITE FIELD TO FILE I

for( n=0; n<narr; n++ )
{
    for( y=0; y<ydim; y++ )
    {
        for( x=0; x<xdim; x++ )
        {
            fprintf( fptr, "%e\n", H[x][y][0][n] );
        }
    }
}

// CLOSE FILE I
fclose( fptr );
9.O Listing of quadpole.c

/* PROGRAM quadpole.c */
/* The details of this program are described in CPL-620. */

#include <stdio.h> /* Load standard i/o routines */
#include <math.h> /* Load standard math routines */
#define narr 3 /* Set number of arrays */
#define xdim 30 /* Set dimensions: 30x30x3 */
#define ydim 30
#define zdim 30
#define xrange 4.0 /* Set length of interval in x */
#define yrange 4.0 /* Set length of interval in y */
#define zrange 4.0 /* Set length of interval in z */
#define x0 -2.0 /* Set starting value of x */
#define y0 -2.0 /* Set starting value of y */
#define z0 -2.0 /* Set starting value of z */

main()
{
    FILE *fptr; /* For file pointer */
    float E[30][30][30][3]; /* For electric field */
    float xf, yf, zf; /* For coordinates */
    float dx, dy, dz; /* For increments */
    float xfm, xfp, yfm, yfp; /* For x+1, x-1, etc. */
    float xfms, xfps, yfms, yfps, zfs; /* For (x+1)^2, (x-1)^2, etc. */
    float rmm, rmp, rpm, rpp; /* For denominators */
    int n, x, y, z; /* For loop control */
    dx = xrange/(xdim - 1.0); /* Set increments */
    dy = yrange/(ydim - 1.0);
    dz = zrange/(zdim - 1.0);

    /***** EVALUATE ELECTRIC FIELD *****/

    for( z=0; z<zdim; z++ )
    {
        zf = z * dz + z0;
        zfs = zf*zf;
        for( y=0; y<ydim; y++ )
        {
            yf = y * dy + y0;
            yfm = yf - 1.0;
            yfp = yf + 1.0;
            yfms = yfm*yfm;
            yfps = yfp*yfp;
            for( x=0; x<xdim; x++ )
            {
                xf = x * dx + x0;
                xfm = xf - 1.0;
                xf = xf + 1.0;
                xfms = xfm*xfm;
xfps = xfp*xfp;
rm = pow(xfms + yfms + zfs, 1.5);
rmp = pow(xfms + yfps + zfs, 1.5);
rpp = pow(xfps + yfms + zfs, 1.5);

E[x][y][z][0] = xfm*(1.0/rmm - 1.0/rmp)  /* Ex */
+ xfp*(1.0/rpp - 1.0/rpm);
E[x][y][z][1] = yfm*(1.0/rmm - 1.0/rpm)  /* Ey */
+ yfp*(1.0/rpp - 1.0/rpm);
E[x][y][z][2] = zf*(1.0/rmm - 1.0/rmp
+ 1.0/rpp - 1.0/rpm);

fptr = fopen( "quadpole_c.dat", "w" );

fclose( fptr );
/* PROGRAM read_irrad.c */

/* This program reads the file irrad_c.dat containing data on the */
/* irradiance produced in the diffraction pattern of a rectangular */
/* aperture. The file contains a single 50x50 array, but that */
/* array is viewed as a four-dimensional array having dimensions */
/* 50x50x1x1. */

#include <stdio.h>
#include <math.h>

main()
{
    FILE *fptr; /* For file pointer */
    char ln1[120], ln2[120], ln3[120]; /* For comments */
    char ln4[120], ln5[120]; /* For dimensions of array */
    int narr, ix, iy, iz; /* For loop indices */
    int n, i, j, k; /* For dimensions of array */
    float I[50][50][1][1]; /* For array */

    fptr = fopen( "irrad_c.dat", "r" ); /* Open file */

    fgets( ln1, 120, fptr ); /* Read comment lines */
    fgets( ln2, 120, fptr );
    fgets( ln3, 120, fptr );
    fgets( ln4, 120, fptr );
    fgets( ln5, 120, fptr );

    fscanf(fptr, "%d%d%d%d", &narr,&ix,&iy,&iz); /* Read dimensions */

    for( n=0; n<narr; n++ ) /* Read data for array */
    {
        for( k=0; k<iz; k++ )
            for( j=0; j<i; j++ )
                for( i=0; i<ix; i++ )
                    fscanf( fptr, "%g", &I[i][j][k][n] );

    }

    fclose( fptr ); /* Close file */

    /* Coding that processes the array as desired would then follow */
    /* at this point in the program. */
}
Chapter 10

Introduction to Numerical Recipes

Note: If your site owns licenses for the Numerical Recipes library, files in this chapter that are components of the Numerical Recipes library can be downloaded from the directory $NRHEAD. Here, as defined in the Local Guide, $NRHEAD must be replaced by the appropriate path for your site. The subdirectories in $NRHEAD are identified in Fig. 10.1.

The Numerical Recipes package contains numerous subroutines implementing a wide variety of numerical algorithms and provides a valuable time-saving starting point for the writing of programs to carry out almost any numerical analysis. Some programming knowledge in FORTRAN or C is prerequisite to successful use of the recipes. In this chapter, we provide basic information about the way the files in the package are stored, and we illustrate how to use those files. Additional applications are described in subsequent chapters. These several discussions, however, are in no way intended to comprise the ultimate document on the usage of the recipes. Full documentation is contained in two source books: Numerical Recipes: The Art of Scientific Computing (hereafter referred to as the main source book) and Numerical Recipes: Example Book (the example source book). These books describe not only the various methods and routines employed by the recipes but also the mathematical theory behind them. A pair of books exists for each of several programming languages. In the second edition, FORTRAN (both 77 and 90), C, and C++ are supported.

10.1 The Numerical Recipes Directory Tree

The hundreds of files that make up the Numerical Recipes package are stored in a directory tree whose head is $NRHEAD. As illustrated in Fig. 10.1, the full tree contains three subdirectories named recipes_f, recipes_c-ansi, and recipes_c-kr, though not all of these subdirectories will be present at every site. Each of these subdirectories in turn contains several (sub)directories, the most significant of which are named recipes, demo, and (for the C directories) include. Further, the directory demo contains several still lower level directories, including data and src. For each language,

---

1 The Numerical Recipes package is a commercial product marketed by Numerical Recipes Software, Cambridge, MA, 02238, and by Cambridge University Press. (See Appendix Z for full contact information.) Its use at any particular site is subject to the provisions of whatever license that site has negotiated with Numerical Recipes Software. The terms of that license are explained in the Local Guide.

2 The names and calling sequences of the recipes are copyrighted by Numerical Recipes Software and are used in this book with permission of Numerical Recipes Software.

3 These reference works are published by Cambridge University Press; the authors are William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery.

4 Other languages, including Pascal, LISP, Modula2, and some versions of BASIC, are still available in the first edition. The third edition, which is not described in this chapter, is the current edition—though it supports only C++.

5 See the Local Guide for the translation of this symbol at your site.
• The directory $NRHEAD/recipes\_language/recipes$ contains all of the recipes themselves. These recipes are described in detail in the main source books.

• The directory $NRHEAD/recipes\_language/demo/src$ contains many demonstration programs, which illustrate how to construct driving programs to use nearly every recipe in the recipes subdirectories. These programs can be used as they stand, or they can be used as a starting point for the writing of customized driving programs. They are described in the example source books.

• The directory $NRHEAD/recipes\_language/demo/data$ contains the data files needed by a few of the demonstration programs.

• The directories $NRHEAD/recipes\_language/include$ in the C libraries contain header and include files that are needed by some of the C recipes.

When use of a numerical recipe is appropriate, we should first peruse the main source book for a recipe accomplishing the desired task. Then, to develop a "feel" for how the recipe is used, we should examine the listing of the corresponding demonstration program. Finally, we should compile and run that demonstration program in its library form. Only then are we ready to adapt the demonstration program (and perhaps the recipe itself) to our specific purposes. In the remainder of this chapter, we illustrate the process by which the library form of a particular recipe and its associated demonstration program can be compiled and run. Later chapters illustrate how to use one or another recipe for more specific numerical analysis.

### 10.3 Using C Recipes

In C, the files in the subdirectory recipes have names with filetype .c. The name of the demonstration program in the subdirectory demo/src is obtained from the name of the associated recipe by prefixing an x (presumably derived from the ‘x’ in ‘example’). Thus, for example, the first recipe found in the main source books—a recipe that calculates the dates of occurrence of the phases of the moon—is called flmoon and is stored in a file named flmoon.c. The associated demonstration file is named xflmoon.c. To use the demonstration program,
1. We copy the program `xflmoon.c` into our default directory with the statement\(^6,7\)

   ```bash
   cp $NRHEAD/recipes_c-ansi/demo/src/xflmoon.c .
   ```

2. We copy the associated recipe `flmoon.c` into our default directory with the statement

   ```bash
   cp $NRHEAD/recipes_c-ansi/recipes/flmoon.c .
   ```

3. We determine whether the program invokes recipes beyond the primary one. One way to make this determination entails compiling and loading the program with the statement

   ```bash
   cc -o xflmoon.xc xflmoon.c flmoon.c
   ```

   If this statement generates no error messages, then no further recipes are needed. More likely, others will be needed; if so, they will be identified in the error messages. In the present case, the file `nr.h`, which is a special header file, is not found. Thus, you must copy it from its location in the public directory structure with the statement

   ```bash
   cp $NRHEAD/recipes_c-ansi/include/nr.h .
   ```

   Then, a repeat compilation with the statement

   ```bash
   cc -o xflmoon.xc xflmoon.c flmoon.c
   ```

   reveals a number of additional “unresolved text symbols”, including `julday`, `caldat`, `nrerror`, `sin`, `floor`, and `ceil`. The first two of these are additional needed recipes, which can be copied with the statements

   ```bash
   cp $NRHEAD/recipes_c-ansi/recipes/julday.c .
   cp $NRHEAD/recipes_c-ansi/recipes/caldat.c .
   ```

   the third is a utility routine contained within a package named `nrutil.c`, which can be copied with the statement

   ```bash
   cp $NRHEAD/recipes_c-ansi/recipes/nrutil.c .
   ```

   while the remaining three are items from the standard C mathematics library `m`, which must therefore be explicitly included in the compile command with the option `-lm`.

   A more gentle way to identify needed additional files entails examining the listings of the program `xflmoon.c` and the recipe `flmoon.c`, noting all non-standard include files and all non-standard procedure calls. In the present case, examining `xflmoon.c` reveals the inclusion of `nr.h` and calls to `julday` and `caldat` while examination of `flmoon.c` reveals invocation of `nrerror` (which is defined in `nrutil.c`), `sin`, `floor`, and `ceil`. Strictly, we should then examine the listings of any subroutines and other entities so identified to see if they in turn invoke still other subroutines. Together, these invocations identify the same spectrum of additional routines as those to which we were led in the first approach.

4. We compile and load the program with the statement\(^8\)

\(^6\)To be specific, both here and throughout this section, we illustrate these steps with the statements that would be used in UNIX to compile, link, and run the program. Other operating systems probably accomplish the same end with different statements. Details will be found in the Local Guide.

\(^7\)We also assume the ANSI version of C is in use. To use the KR version, simply replace the designation of the language directory with `recipes_kr`. Consult your Local Guide for the version available at your site.

\(^8\)This statement may generate a warning about conflicting data types for one or more variables. Warnings, however, do not prevent successful coomilation.
cc -o xflmoon.xc xflmoon.c flmoon.c julday.c caldat.c nrutil.c -lm

Except that the component -lm should occur last in this list, the order of the filenames is otherwise irrelevant. Note, incidentally, that the cc command will leave a succession of object files (filetype .o) in our directory. These files may be deleted once the executable program has been created.

5. We run the program with the statement

   ./xflmoon.xc

The program asks for a date (and prescribes the format in which the date is to be entered) and then prints out the dates of the next several occurrences of each of the four phases of the moon.

This discussion is, of course, purely exemplary of the procedure we will have to invoke to use a particular numerical recipe in its library form. More often, we will need to use the demonstration program as a template for creating a driving program better suited to our specific purposes. Occasionally, we may have to edit the recipes themselves. Illustrations of these operations will be found in Sections 11.15.2, 13.14.2, and 14.15.2.

10.4 Exercises

10.3. The recipe julday.c calculates and prints out the Julian day number corresponding to a date supplied to it by a suitable driving program. (a) Read what the main and example source books have to say about this recipe and its associated demonstration program. (b) Retrieve the demonstration program xjulday.c, the primary recipe julday.c, and any necessary additional files from the Numerical Recipes library, and examine the listing of the demonstration program. (c) Compile and run the library form of the program. (d) In particular, determine how old you are in days by determining the Julian day number of your birthday and of today. Note: The file dates1.dat needed by the program can be copied from the directory $NRHEAD/recipes_c-ansi/demo/data.

10.4. The recipe avevar.c calculates the mean and variance of a data set supplied to it by a suitable driving program. (a) Read what the main and example source books have to say about this recipe and its associated demonstration program. (b) Retrieve the demonstration program xavevar.c, the primary recipe avevar.c, and any necessary additional files from the Numerical Recipes library, and examine the listing of the demonstration program. (c) Compile and run the library form of the program. (d) Finally, edit the driving program so that it supplies a different set of data of your choice to the recipe. Then compile and run your edited program.

---

9Again, we are using UNIX-style statements. In some operating systems, the characters ./ preceding the name of the program to be executed may not be necessary.
Chapter 11

Solving Ordinary Differential Equations

Many fundamental laws of physics relate the rate at which the physical properties of a system change to the properties themselves. These physical laws lead inevitably to differential equations satisfied by the quantities describing the system. While some of these equations admit closed form, symbolic solutions, most can be solved only through numerical approximation. We begin this chapter by identifying several physical situations, the full addressing of which requires us to solve an ordinary differential equation (ODE) or a coupled set of such equations. Then we illustrate how to use symbolic algebra systems to approach those that can be solved analytically, describe a few of many available numerical algorithms (with attention to their accuracy), and—finally—describe ways to solve representative ODEs using a variety of numerical approaches and computational tools.

Differential equations fall into many, sometimes overlapping, categories. We limit ourselves in this chapter to *ordinary* differential equations, which involve only one independent variable. Most equations of interest in physics are first-order (containing no derivatives higher than the first) or second-order (containing no derivatives higher than the second), but occasionally higher order equations may arise. Whatever their order, these equations may be linear (each term depending on the dependent variable only through either its first power or the first power of one of its derivatives) or non-linear (at least one term violating the constraint in the previous parentheses). They may be homogeneous (no term free of the dependent variable) or inhomogeneous (at least one term free of the dependent variable). The coefficients may be constant or may depend on the independent variable. Most will contain parameters characterizing the system of interest, though recasting the original equations in dimensionless form may reduce the number of distinct parameters—or even eliminate them altogether. We may be confronted with a single equation (one dependent variable) or with a system of equations (two or more dependent variables), and the members of the system may be coupled (more than one of the dependent variables appearing in at least one of the equations) or decoupled (no member of the system containing more than one of the dependent variables).

For complete statement of a problem, the applicable ODEs must be supplemented with auxiliary conditions, the number of which equals the sum of the orders of the equations at hand. A single first-order equation requires one condition: stipulation of the value of the dependent variable at a specific value of the independent variable is sufficient to select a unique solution from the family of solutions defined by the differential equation alone. A single second-order equation requires two conditions but, in this case, we have some choices. We might, for example, stipulate the value of the dependent variable and the value of its first derivative at a single value of the independent variable, e.g., position and velocity at an initial time. In that case, we would be dealing with an *initial value problem* (IVP). Alternatively, we might stipulate the value of the dependent variable at each of two different values of the independent variable, e.g, displacement of a string at each of its two ends. In that case, we would be dealing with a *boundary value problem* (BVP).
The approach to solving a particular ODE or system of ODEs may well be dictated by the category into which the equation or equations fall. The approach will also be influenced by whether we are dealing with an IVP or a BVP. The examples chosen for this chapter illustrate several of these situations.

11.1 Sample Problems

In this section, we identify several physical contexts that lead to differential equations, and we determine the specific differential equation arising in each case. Solutions to the resulting equations by a variety of symbolic and numerical means will be explored in the remainder of this chapter.

In most cases the statement of the problem of interest will contain several constants or parameters. Some reside in the ODE itself while others reside in the initial or boundary conditions. The presence of such constants gives rise to two complications. First, a system of equations containing many constants is much more difficult to explore than a system containing only a few constants. Second, in some cases, the values of the constants will be either very large (e.g., planetary distances or masses) or very small (e.g., atomic distances or masses). In these cases, finding appropriate initial conditions can be difficult. Additionally, numbers of these magnitudes can potentially cause floating point overflow or underflow. Frequently, both of these complications can be made less severe by casting the differential equation(s) and associated initial or boundary conditions in dimensionless form. To accomplish that objective, we begin by choosing judicious, non-standard units in terms of which to express the independent and dependent variables. Then, we rescale these variables to express them in the chosen units. Sometimes, all parameters in the equations and the initial or boundary conditions will disappear; more often, a small number of (dimensionless) combinations of parameters will remain. In any case, the recast problem is almost certain to be simpler than the original problem, partly because the solution depends on fewer “real” parameters and partly because the significant values of all quantities are likely to have order of magnitude one. Anticipating that dimensionless presentations will facilitate some of our subsequent solutions, we shall conclude several of the subsections in this section by illustrating how the strategy described in this paragraph would be implemented for the equations in those subsections.

11.1.1 Projectile in a Viscous Medium

The projectile shown in Fig. 11.1 moves in a viscous medium near the surface of the earth. It experiences two forces, the gravitational attraction of the earth $-mg$ and the viscous force $F_v$ from the medium in which it moves. The former is directed downward and the latter is directed...

\footnote{We choose a coordinate system in which positive $z$ is directed upward, and we take $m$ and $g$ to be positive.}
11.1. SAMPLE PROBLEMS

opposite to the velocity \( \mathbf{v} \). Usually, the magnitude of the viscous force is a function of the speed with which the projectile moves, symbolically \( |\mathbf{F}_v| = f(|\mathbf{v}|) \). In general, if the projectile has mass \( m \), Newton’s second law yields the equation of motion

\[
m \frac{d^2 \mathbf{r}}{dt^2} = -mg \, \hat{k} + \mathbf{F}_v(|\mathbf{v}|) = -mg \, \hat{k} - \frac{f(|\mathbf{v}|)}{|\mathbf{v}|} \mathbf{v}
\]

which we are to solve subject to the general initial conditions,

\[
\mathbf{r}(0) = \mathbf{r}_0 = x_0 \, \hat{i} + y_0 \, \hat{j} + z_0 \, \hat{k} ; \quad \mathbf{v}(0) = \mathbf{v}_0 = v_{x0} \, \hat{i} + v_{y0} \, \hat{j} + v_{z0} \, \hat{k}
\]

Of course, we also need to know the precise dependence of the function \( f(|\mathbf{v}|) \) on the speed of the projectile. In the simplest case (when the speed of the projectile is small enough), \( f \) is simply linearly proportional to that speed, \( f(|\mathbf{v}|) = b \, |\mathbf{v}| \) (\( b \) a positive constant), and the equation of motion reduces to

\[
m \frac{d^2 \mathbf{r}}{dt^2} = -mg \, \hat{k} - b \mathbf{v} = -mg \, \hat{k} - b \frac{d \mathbf{r}}{dt}
\]

or, in component form, to

\[
m \frac{d^2 x}{dt^2} = -b \frac{dx}{dt} ; \quad m \frac{d^2 y}{dt^2} = -b \frac{dy}{dt} ; \quad m \frac{d^2 z}{dt^2} = -mg - b \frac{dz}{dt}
\]

This system of equations is uncoupled, since each of the three independent variables \( x, y, \) and \( z \) satisfies its own private equation that does not involve either of the other variables. They are second-order and linear, and the coefficients are constant. The first two are homogeneous and, because of the term \(-mg\), the third is inhomogeneous. The equations involve the parameters \( m, b, \) and \( g \), and their solutions will depend on these parameters and on the six initial values in Eq. (11.2).

When the speed is too large for a linear approximation to the viscous damping, we can sometimes take the viscous force to be proportional instead to the square of the speed, \( f = b|\mathbf{v}|^2 \) (\( b \) a positive constant, though not the same constant as in the previous paragraph). This time, the equation of motion reduces to

\[
m \frac{d^2 \mathbf{r}}{dt^2} = -mg \, \hat{k} - b|\mathbf{v}| \mathbf{v} = -mg \, \hat{k} - b \frac{d \mathbf{r}}{dt} \sqrt{\left( \frac{dx}{dt} \right)^2 + \left( \frac{dy}{dt} \right)^2 + \left( \frac{dz}{dt} \right)^2}
\]

or, in component form, to

\[
m \begin{bmatrix} \frac{d^2 x}{dt^2}, \frac{d^2 y}{dt^2}, \frac{d^2 z}{dt^2} \end{bmatrix} = [0, 0, -mg] - b \begin{bmatrix} \frac{dx}{dt}, \frac{dy}{dt}, \frac{dz}{dt} \end{bmatrix} \sqrt{\left( \frac{dx}{dt} \right)^2 + \left( \frac{dy}{dt} \right)^2 + \left( \frac{dz}{dt} \right)^2}
\]

This system of equations is second-order and distinctly non-linear, and its members are coupled because each of the equations involves all three of the dependent variables. Even if the motion occurs in only the vertical dimension (projectile tossed straight up or, simply, released from rest and allowed to drop), the equation, which then is

\[
m \frac{d^2 z}{dt^2} = -mg - b \frac{dz}{dt} \frac{dz}{dt}
\]

is complicated by the absolute value in the viscous term. These equations involve the parameters \( m, b, \) and \( g \), and their solutions will depend on these parameters and on the six initial values in Eq. (11.2).

To cast these equations in a dimensionless form, we would start by choosing a unit of length, say \( a \)—which might be chosen arbitrarily, chosen to be one of the initial positions, or merely symbolized initially and chosen later to simplify the equations as we converge on their dimensionless form. Then, dividing by \( m \) and rescaling the dimensional coordinates by introducing the dimensionless
coordinates \( \bar{x} = x/a, \bar{y} = y/a \), and \( \bar{z} = z/a \) (i.e., \( \bar{\mathbf{r}} = \mathbf{r}/a \)), we find that Eqs. (11.3) and (11.5) become

\[
\frac{d^2 \bar{r}}{dt^2} = -\frac{g}{a} k - \frac{b}{m} \frac{d\bar{r}}{dt}
\]

and

\[
\frac{d^2 \bar{r}}{dt^2} = -\frac{g}{a} k - \frac{b}{m} \frac{\bar{r}}{a} \sqrt{\left(\frac{d\bar{x}}{dt}\right)^2 + \left(\frac{d\bar{y}}{dt}\right)^2 + \left(\frac{d\bar{z}}{dt}\right)^2}
\]

respectively. Next, we recognize that \( \sqrt{g/a} \) is dimensionally a frequency.\(^2\) Hence, the variable \( \bar{t} = t \sqrt{g/a} \) provides a suitable rescaling of the independent variable. In terms of \( \bar{t} \), Eqs. (11.8) and (11.9) become

\[
\frac{d^2 \bar{r}}{d\bar{t}^2} = -\hat{k} - \frac{\hat{b}}{\hat{m}} \frac{d\bar{r}}{d\bar{t}}
\]

and

\[
\frac{d^2 \bar{r}}{d\bar{t}^2} = -\hat{k} - \frac{b}{a} \frac{\bar{r}}{a} \sqrt{\left(\frac{d\bar{x}}{d\bar{t}}\right)^2 + \left(\frac{d\bar{y}}{d\bar{t}}\right)^2 + \left(\frac{d\bar{z}}{d\bar{t}}\right)^2}
\]

Finally, we introduce the symbol \( \bar{\beta} \) for the dimensionless quantity \( b\sqrt{a/g}/m \) in the first of these equations and for the dimensionless quantity \( ba/m \) in the second.\(^3\) In the end, the equations in dimensionless form are

\[
\frac{d^2 \bar{r}}{d\bar{t}^2} = -\hat{k} - \bar{\beta} \frac{d\bar{r}}{d\bar{t}}
\]

and

\[
\frac{d^2 \bar{r}}{d\bar{t}^2} = -\hat{k} - \bar{\beta} \frac{\bar{r}}{a} \sqrt{\left(\frac{d\bar{x}}{d\bar{t}}\right)^2 + \left(\frac{d\bar{y}}{d\bar{t}}\right)^2 + \left(\frac{d\bar{z}}{d\bar{t}}\right)^2}
\]

These equations are, of course, to be solved subject to the dimensionless initial conditions

\[
\bar{r}(0) = \frac{r_0}{a} \quad ; \quad \bar{v}(0) = \frac{d\bar{r}}{d\bar{t}}(0) = \frac{d(r/a)}{d(t\sqrt{g/a})}(0) = \frac{v_0}{\sqrt{ga}}
\]

from which we conclude that dimensionless velocities are measured in units of \( \sqrt{ga} \).

In truth, however, these particular equations are not really second-order equations. They can be reduced to first-order equations by focusing attention on the components \( [v_x, v_y, v_z] \) of the velocity as the dependent variables. In those terms, the three members of Eq. (11.4) become

\[
m \frac{dv_x}{dt} = -b v_x \quad ; \quad m \frac{dv_y}{dt} = -b v_y \quad ; \quad m \frac{dv_z}{dt} = -mg - bv_z
\]

the three members of Eq. (11.6) become

\[
m \left[ \frac{dv_x}{dt}, \frac{dv_y}{dt}, \frac{dv_z}{dt} \right] = [0, 0, -mg] - b [v_x, v_y, v_z] \sqrt{v_x^2 + v_y^2 + v_z^2}
\]

and Eq. (11.7) becomes

\[
m \frac{dv_z}{dt} = -mg - bv_z |v_z|
\]

Once these first-order equations have been solved, the components of the position vector can then be found by solving the differential equations

\[
\frac{dx}{dt} = v_x \quad ; \quad \frac{dy}{dt} = v_y \quad ; \quad \frac{dz}{dt} = v_z
\]

\(^2\)Using \([\ldots]\) to indicate “the dimensions of . . .”, we argue \([g] = \text{length}/\text{time}^2\), \([a] = \text{length} \implies [g/a] = \text{time}^{-2} \implies [\sqrt{g/a}] = \text{time}^{-1}\).

\(^3\)Remember that \( b \) is not the same in the two instances.

\(^4\)Here (and in all subsequent cases where a chosen unit is identified but not checked), you should take a moment to verify that the identified unit has the proper dimensions.
By the time we have reached this point, however, we will have explicit knowledge of \( v_x, v_y, \) and \( v_z \) as functions of \( t \), so solving Eq. (11.18) is equivalent to straightforward evaluation of an integral. (See Chapter 13.) Recasting the equations of this paragraph in dimensionless form is left as an exercise.

### 11.1.2 Chain Radioactive Decay

A wholly different context in which systems of ODEs arise lies in radioactive decay. The fundamental law asserts that a sample of a particular radioisotope decays at a rate proportional to the quantity of (undecayed) material present in the sample. Thus, for the decay chain shown in Fig. 11.2, we would write the system of three differential equations

\[
\frac{dA}{dt} = -k_A A ; \quad \frac{dB}{dt} = k_A A - k_B B ; \quad \frac{dC}{dt} = k_B B
\]

\[(11.19)\]

where \( k_A \) and \( k_B \) are decay constants (parameters); \( A(t), B(t), \) and \( C(t) \) are the number of nuclei of each species present; and nucleus \( C \) is assumed to be stable. These equations are linear, first-order, and homogeneous, and they have constant coefficients. They are, however, coupled, since each of the second and third of them involves two of the dependent variables. They also support a conservation law: adding the three equations yields that

\[
\frac{d}{dt} \left( A + B + C \right) = 0 \Rightarrow A + B + C = \text{constant} \quad (11.20)
\]

As always we, of course, need initial values, e.g., \( A(0) = A_0, B(0) = 0, \) and \( C(0) = 0, \) before the differential equations have a unique solution, and that solution will depend on the parameters \( k_A \) and \( k_B \) and on the three initial values.

To cast these equations in a dimensionless form, we choose a reference amount—here, conveniently, \( A_0 \), the initial amount of \( A \)—as the unit for measuring the quantities of \( A, B, \) and \( C \). Then, we rescale the values for these quantities by introducing the dimensionless variables \( \bar{A} = A/A_0, \bar{B} = B/A_0, \) and \( \bar{C} = C/A_0 \). Next, dividing the equations by \( k_A A_0 \) and introducing the dimensionless quantities \( \bar{t} = k_A t \) and \( \bar{k} = k_B/k_A \), we conclude that

\[
\frac{d\bar{A}}{d\bar{t}} = -\bar{A} ; \quad \frac{d\bar{B}}{d\bar{t}} = \bar{A} - \bar{k} \bar{B} ; \quad \frac{d\bar{C}}{d\bar{t}} = \bar{k} \bar{B}
\]

\[(11.21)\]

with the initial conditions \( \bar{A}(0) = 1, \bar{B}(0) = 0, \) and \( \bar{C}(0) = 0, \) where we now regard the dependent variables as functions of the dimensionless time \( \bar{t} \). In short, we discover that this problem possesses only one “real” parameter \( \bar{k} \). Only the ratio of the rate constants conveys any significant distinction among different realizations of this decay. Everything else is simply a matter of scaling, either on the time variable or on the dependent variables as a group. The essential physics is both easier to explore and easier to comprehend when the problem is viewed from a dimensionless perspective.

---

5 We shall later see that conservation laws can sometimes prove valuable in assessing the accuracy of solutions.
11.1.3 Exponential and Logistic Growth

An important illustration of a non-linear first-order equation occurs in population biology, where—in the absence of predation—the population of a species grows at a rate proportional to that population until the population becomes so large that individual organisms compete significantly with one another for space and/or food. In a simple model, the effects of competition are proportional to the likelihood that one organism will encounter another—a likelihood that is proportional to the square of the population. Thus, a population subject to both effects will evolve in accordance with the first-order, non-linear equation

\[ \frac{dN}{dt} = kN \left(1 - \frac{N}{N_c}\right) \]  

(11.22)

where \(N(t)\) is the population, \(k\) is the growth rate, and \(N_c\), which is the value of \(N\) at which its rate of growth becomes zero, is the carrying capacity of the environment. If, in particular, \(N(t)\) ever equals \(N_c\), then \(dN/dt = 0\) and \(N\) ceases to change; the population will attain an equilibrium, which it maintains forever after. We must, of course, know the initial population \(N(0) = N_0\) before a complete solution to this equation can be found, and that solution will depend on the parameters \(k\) and \(N_c\) and on the initial value \(N_0\).

Two different terms are used to label the solutions to Eq. (11.22). If \(N_c\) is infinite (or, more realistically, \(N(t) \ll N_c\)), then \(dN/dt = 0\) and the resulting growth is said to be exponential, though the growth will actually be a decay if \(k < 0\). If, on the other hand, \(N(t)\) is not small compared to \(N_c\), both terms are important, the exponential growth of the first case reaches a ceiling, and the growth is said to be logistic.  

The dimensionless version of this equation is quickly found. We choose \(N_c\) as the reference population, introduce the dimensionless population \(\bar{N} = N/N_c\) and the dimensionless time \(\bar{t} = kt\), and find that the equation and initial condition reduce to

\[ \frac{d\bar{N}}{d\bar{t}} = \bar{N}(1 - \bar{N}) \; ; \; \bar{N}(0) = \frac{N_0}{N_c} \]  

(11.23)

All parameters disappear from the differential equation but the initial population—now measured in units of the carrying capacity—remains as a single parameter in the problem.

11.1.4 Forced, Driven, Damped Harmonic Oscillation

A particularly important, second-order differential equation arises in several contexts. Suppose, for example, as shown in Fig. 11.3, an object of mass \(m\) moves on a horizontal, frictionless surface under the action of forces applied by a Hooke’s law spring of constant \(k\), a viscous shock absorber of damping constant \(b\), and an externally applied time-dependent force \(F(t)\). Newton’s second law leads us to write the linear, second-order, constant coefficient equation of motion governing this system as

\[ m \frac{d^2x}{dt^2} = -kx - b \frac{dx}{dt} + F(t) \]  

(11.24)

Here, \(x\) is measured from the position of the object when the spring is neither stretched nor compressed. This equation is inhomogeneous if \(F \neq 0\) and homogeneous if \(F = 0\). As always, we require initial conditions, which will have the general form

\[ x(0) = x_0 \; ; \; v(0) = v_0 \]  

(11.25)

before the problem is fully stated.

---

6Logistic growth is also sometimes said to follow a sigmoid curve because of the stylized ‘S’ shape that solutions exhibit when the initial population is much smaller than the carrying capacity. Graphs of this shape will be found in subsequent sections of this chapter.
To cast this equation in dimensionless form, we introduce a unit of length, say $\ell$, rescale position with the expression $x = x/\ell$, introduce a unit of time, say $\tau$, rescale the physical time with the expressions $t = t/\tau$, and thereby transform Eq. (11.24) to

$$\frac{d^2 x}{dt^2} = -\frac{\tau^2 k}{m} x - \frac{\tau b}{m} \frac{dx}{dt} + \tau^2 \frac{F(\tau t)}{m \ell}$$

(11.26)

The term on the left in this equation is now dimensionless. Thus, all terms on the right must be dimensionless as well. In particular, the combination $\tau^2 k/m$ must be dimensionless. Remember, however, that $\tau$ is at the moment merely a symbol; we have not yet made a well defined choice for the unit of time, and we are free to choose $\tau$ any way we like. Clearly, the choice $\tau^2 k/m = 1$ or $\tau = \sqrt{m/k}$ is judicious. With this choice, the equation of motion becomes

$$\frac{d^2 \bar{x}}{d\bar{t}^2} = -\bar{x} - \frac{b}{\sqrt{mk}} \frac{d\bar{x}}{d\bar{t}} + \bar{F}(\bar{t})$$

(11.27)

where, in the final form, we have set $\bar{\beta} = b/\sqrt{mk}$ and $\bar{F}(\bar{t}) = F(\tau t)/k \ell$. The single, dimensionless parameter $\bar{\beta}$ contains the essential influence of the three parameters $m$, $b$, and $k$ once differences attributable to scaling have been removed. The dimensionless force $\bar{F}$ expresses the physical force in units of $k \ell$, which—note—is the force that the spring would exert if extended by the chosen unit of length!

We must, of course, also translate the initial conditions of Eq. (11.25) into dimensionless form, finding that

$$\bar{x}(0) = \frac{\bar{x}}{\ell} ; \quad \bar{v}(0) = \frac{d\bar{x}}{d\bar{t}}(0) = \frac{d(x/\ell)}{d(t/\tau)} = \frac{v_0}{\ell/\tau}$$

(11.28)

and we conclude that dimensionless velocities will be measured in units of $\ell/\tau$, which is the speed of an object that moves the reference distance $\ell$ in the reference time $\tau$.

Alternatively (and, in some approaches to solution, necessarily), we would recast this single second-order differential equation as a pair of first-order equations, either

$$\frac{dx}{dt} = v ; \quad m \frac{dv}{dt} = -kx - bv + F(t)$$

(11.29)

in the original dimensional form, or

$$\frac{d\bar{x}}{d\bar{t}} = \bar{v} ; \quad \frac{d\bar{v}}{d\bar{t}} = -\bar{x} - \bar{\beta} \bar{v} + \bar{F}(\bar{t})$$

(11.30)

in dimensionless form. All dimensionless quantities are those introduced earlier in this section.
11.1.5 An LRC Resonant Circuit

An equation mathematically identical in form to Eqs. (11.29) and (11.30) arises in the description of a series RLC circuit excited by a signal generator, as shown in Fig. 11.4. If we take positive current \( i(t) \) to flow clockwise and understand that \( q(t) \) represents the charge on the left plate of the capacitor, then Kirchhoff’s loop equation and the properties of the several components lead to the equation

\[
V(t) - L \frac{di}{dt} - \frac{q}{C} - iR = 0 \quad \Rightarrow \quad L \frac{di}{dt} + iR + \frac{q}{C} = V(t)
\]  

(11.31)

As it stands, this equation looks to be first order but it involves two variables \( i(t) \) and \( q(t) \). We can complete the statement of a problem having a unique solution by recognizing the relationship

\[
i = \frac{dq}{dt}
\]

(11.32)

between \( i \) and \( q \) and supplementing what is now a pair of coupled first-order, linear, constant coefficient, inhomogeneous equations with the general initial conditions

\[
q(0) = q_0 ; \quad \frac{dq}{dt}(0) = i(0) = i_0
\]

(11.33)

Alternatively, we could substitute Eq. (11.32) into Eq. (11.31) to find the equivalent, single, second-order, linear, constant coefficient, inhomogeneous equation

\[
L \frac{d^2q}{dt^2} + R \frac{dq}{dt} + \frac{q}{C} = V(t)
\]

(11.34)

The recasting of these equations in dimensionless form is left as an exercise.

Adopting the correspondences, \( q \leftrightarrow x \), \( L \leftrightarrow m \), \( R \leftrightarrow b \), \( 1/C \leftrightarrow k \), \( V(t) \leftrightarrow F(t) \), and \( t \leftrightarrow t \), we can turn Eq. (11.34) into Eq. (11.24). Thus, mathematically, the driven, damped mechanical oscillator and the RLC circuit exhibit exactly analogous behavior, and the behavior of the RLC circuit simulates the behavior of the mechanical oscillator.\(^7\)

11.1.6 Coupled Oscillators

Consider next the system shown in Fig. 11.5 consisting of two objects of equal mass \( m \) connected along a line, each to a fixed wall by springs of constant \( k \) and each to the other by a coupling spring

\(^7\)Years ago, when digital computers were not as fast as they have come to be and smooth graphical output from a digital computer was unheard of, correspondences such as this one were the basis of the analog computer, whereon we could easily set up electronic circuits whose behavior simulated the behavior of more expensive mechanical systems. With an analog computer, we could learn about mechanical systems by observing the real-time variation of the voltages and currents at various points in an analogous electronic circuit.
of constant $k'$. Let the (horizontal) surface on which these objects slide be frictionless, and let $x_1(t)$ and $x_2(t)$ be the displacement of each object from its equilibrium point. Then, Newton’s second law yields

$$m \frac{d^2x_1}{dt^2} = -kx_1 + k'(x_2 - x_1) ; \quad m \frac{d^2x_2}{dt^2} = -kx_2 - k'(x_2 - x_1)$$

(11.35)

for the equations of motion. To reduce the number of parameters, however, we recast these equations in dimensionless form by selecting a unit of length $a$, dividing the equations by $a$, setting $x_i/a = \bar{x}_i$, introducing a dimensionless time variable $\tilde{t} = \omega t$, where $\omega = \sqrt{k/m}$, and setting $\kappa = k'/k$. The equations then become

$$\frac{d^2\bar{x}_1}{d\tilde{t}^2} = -\bar{x}_1 + \kappa (\bar{x}_2 - \bar{x}_1) ; \quad \frac{d^2\bar{x}_2}{d\tilde{t}^2} = -\bar{x}_2 - \kappa (\bar{x}_2 - \bar{x}_1)$$

(11.36)

Initial conditions such as

$$\bar{x}_1(0) = \bar{x}_{10} ; \quad \frac{d\bar{x}_1}{d\tilde{t}}(0) = \bar{v}_{10} ; \quad \bar{x}_2(0) = \bar{x}_{20} ; \quad \frac{d\bar{x}_2}{d\tilde{t}}(0) = \bar{v}_{20}$$

(11.37)

complete the statement of the problem—a problem that involves a pair of coupled, linear, second-order, constant coefficient, homogeneous differential equations containing one internal parameter $\kappa$.

### 11.1.7 Motion under Central Forces

Consider next an object of mass $m$ moving in the $xy$ plane under the action of a central force$^8$ $\mathbf{F}$, as shown in Fig. 11.6. According to Newton’s second law, the position vector $\mathbf{r}$ of this object satisfies

$^8$A central force is one whose direction is always away from or towards a fixed point—the force center—and whose magnitude depends only on the distance from that point.
the differential equation
\[ m \frac{d^2 \mathbf{r}}{dt^2} = \mathbf{F} = f(r) \hat{r} \] (11.38)
where \( f(r) > 0 \) corresponds to a repulsive force and \( f(r) < 0 \) corresponds to an attractive force. Two articulations of this equation are in order. A more specific expression in polar coordinates is treated in an exercise. Here, we extract its components in Cartesian coordinates \((x,y)\). Since
\[ r^2 = x^2 + y^2, \quad \mathbf{r} = x\hat{i} + y\hat{j}, \quad \hat{r} = \frac{x\hat{i} + y\hat{j}}{\sqrt{x^2 + y^2}} \] (11.39)
the Cartesian components of Eq. (11.38) are
\[ m \frac{d^2 x}{dt^2} = f\left(\sqrt{x^2 + y^2}\right) \frac{x}{\sqrt{x^2 + y^2}}; \quad m \frac{d^2 y}{dt^2} = f\left(\sqrt{x^2 + y^2}\right) \frac{y}{\sqrt{x^2 + y^2}} \] (11.40)
We have arrived at a pair of second-order, non-linear, coupled differential equations to be solved subject to general initial conditions of the form
\[ x(0) = x_0; \quad \frac{dx}{dt}(0) = v_x(0) = v_{x0}; \quad y(0) = y_0; \quad \frac{dy}{dt}(0) = v_y(0) = v_{y0} \] (11.41)
Everything discussed so far in this subsection applies to all central forces regardless of the specific form or sign of \( f(r) \). To set a more specific problem, let us narrow our purview to the planetary problem,9 in which a planet of mass \( m \) orbits a central sun of mass \( M \). When \( M \gg m \), as is often the case, the sun does not move appreciably under the action of the gravitational force exerted on it by the planet. Thus, we can treat the gravitational force on the planet as originating in a fixed force center, in which case
\[ f(r) = -\frac{GmM}{r^2} = -\frac{GmM}{x^2 + y^2} \] (11.42)
where \( G \) is the universal gravitational constant. With this specific force, the members of Eq. (11.40) become
\[ \frac{d^2 x}{dt^2} = -\frac{GM}{(x^2 + y^2)^{3/2}} x; \quad \frac{d^2 y}{dt^2} = -\frac{GM}{(x^2 + y^2)^{3/2}} y \] (11.43)
Particularly in the context of this problem, casting the fundamental equations in dimensionless form is prudent. We begin by choosing a reference length, symbolized by \( \ell \). Then, we express all distances in the equations as multiples of this chosen reference by introducing the variables
\[ \xi = \frac{x}{\ell}; \quad \eta = \frac{y}{\ell}; \quad r = \frac{r}{\ell} \] (11.44)
With this change, Eq. (11.43) becomes
\[ \frac{d^2 \xi}{d\bar{t}^2} = -\frac{GM}{\ell^3 \left(\xi^2 + \eta^2\right)^{3/2}} \xi; \quad \frac{d^2 \eta}{d\bar{t}^2} = -\frac{GM}{\ell^3 \left(\xi^2 + \eta^2\right)^{3/2}} \eta \] (11.45)
Finally, we introduce the dimensionless time variable \( \bar{t} = t \sqrt{GM/\ell^3} \) to find that
\[ \frac{d^2 \xi}{d\bar{t}^2} = -\frac{\xi}{\left(\xi^2 + \eta^2\right)^{3/2}}; \quad \frac{d^2 \eta}{d\bar{t}^2} = -\frac{\eta}{\left(\xi^2 + \eta^2\right)^{3/2}} \] (11.46)
Interestingly, in Cartesian coordinates, all parameters have disappeared from the equations.

9Remember, too, that the problem of a charged particle moving in the field of another charged particle is mathematically identical to the planetary problem. (See exercises.)
Translation of the initial conditions into dimensionless form is easier than translation of the differential equations. The positions, of course, become
\[ x(0) = \frac{x_0}{\ell} ; \quad y(0) = \frac{y_0}{\ell} \] (11.47)

To translate the velocities, we argue that
\[ \frac{dx}{dt}(0) = \frac{d(x/\ell)}{d(t \sqrt{GM/\ell^3})}(0) = \frac{v_x}{\ell \sqrt{GM/\ell}} ; \quad \frac{dy}{dt}(0) = \frac{d(y/\ell)}{d(t \sqrt{GM/\ell^3})}(0) = \frac{v_y}{\ell \sqrt{GM/\ell}} \] (11.48)

We conclude that dimensionless velocities in the present context are measured in units of \( \sqrt{GM/\ell} \).

Before leaving this important problem, we note two additional features. First, if the planet happens to be moving at a distance \( r_{circ} \) from the sun with a velocity of magnitude \( v_{circ} \) directed perpendicular to the radius line, then the planet will move in a circular orbit if its speed and radius are related so that the centripetal force \( m v_{circ}^2 / r_{circ} \) needed for circular motion is exactly provided by the gravitational attraction \( GMm/r_{circ}^2 \) of the sun. That is, the orbit will be circular if
\[ \frac{m v_{circ}^2}{r_{circ}} = \frac{GMm}{r_{circ}^2} = \frac{GM}{r_{circ}} \] (11.49)

Translated into dimensionless form, this special relationship becomes
\[ \left( \sqrt{\frac{GM}{\ell}} v_{circ} \right)^2 = \frac{GM}{\ell r_{circ}} = \frac{1}{r_{circ}} \] (11.50)

In the dimensionless units we have chosen, the relationship between speed and radius for a circular orbit involves no dimensional constants. This simple case provides us with a specific known motion against which we can later test numerical solutions for the planetary problem.

Second, the planetary problem admits two conservation laws, each of which may be valuable in assessing the accuracy of numerically generated solutions. In a dimensional presentation, conservation of energy yields that
\[ E = \frac{1}{2} m \left( v_x^2 + v_y^2 \right) - \frac{GMm}{\sqrt{x^2 + y^2}} = \text{constant} \] (11.51)

though the mass \( m \) can be omitted from the expression if only the constancy of \( E \) (and not its actual value) is to be examined. Similarly, in a dimensional presentation, conservation of angular momentum yields that
\[ L = m \left( x v_y - y v_x \right) = \text{constant} \] (11.52)

where, again, \( m \) can be omitted if only the constancy of \( L \) is to be assessed. Recasting these expressions in dimensionless form (and omitting overall multiplying constants), we find alternatively that
\[ \frac{1}{2} \left( \overline{v}_x^2 + \overline{v}_y^2 \right) - \frac{1}{\sqrt{\overline{x}^2 + \overline{y}^2}} = \text{constant} ; \quad \overline{x} v_y - \overline{y} v_x = \text{constant} \] (11.53)

11.1.8 Standing Waves in a String

When a wave propagates in a flexible string, the (transverse) displacement \( u(x, t) \) at time \( t \) of the element of the string nominally at coordinate \( x \) satisfies the wave equation
\[ \frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2} \] (11.54)
where \( c \) is the speed of propagation of the wave along the string. In the special case that the motion of each element is sinusoidal with frequency \( \omega \), we can write
\[
u(x, t) = f(x) \cos(\omega t)
\] (11.55)

and find, on substitution into the wave equation, that \( f(x) \), which gives the amplitude of the sinusoidal motion of the element nominally at \( x \), satisfies
\[
\frac{d^2 f}{dx^2} + k^2 f = 0
\] (11.56)

where \( k^2 = \omega^2 / c^2 \). If, finally, the string is firmly tied down at two points, say \( x = 0 \) and \( x = \ell \), then this second-order, homogeneous, constant-coefficient, differential equation must be solved subject to the boundary conditions
\[
f(0) = 0 \quad ; \quad f(\ell) = 0
\] (11.57)

and we conclude that waves in a string fixed at two points are described by a boundary value problem. Ultimately, we shall be able to find acceptable solutions only for a discrete set of special values of \( k \).

### 11.1.9 The Schrödinger Equation in One Dimension

A quantum mechanical particle of mass \( m \) having definite energy \( E \) and confined in one dimension \( x \) by a potential energy \( V(x) \) is described by a wave function \( \psi(x) \) that satisfies the one-dimensional time-independent Schrödinger equation
\[
-\frac{\hbar^2}{2m} \frac{d^2 \psi}{dx^2} + V(x) \psi = E \psi \quad \implies \quad \frac{d^2 \psi}{dx^2} + \frac{2m}{\hbar^2} \left( E - V(x) \right) \psi = 0
\] (11.58)

where \( \hbar \) is Planck’s constant divided by \( 2\pi \). We anticipate subsequent exploration of two specific cases. Suppose, for example, that the potential energy is infinite except in the interval \( -\ell \leq x \leq +\ell \), in which interval it is zero, i.e., suppose the particle is confined in an infinitely deep potential well that extends over the specified interval. Then, the wave function describing this particle must satisfy the Schrödinger equation with \( V = 0 \) inside this interval, and it must go to zero at each end of the interval. We seek solutions to the boundary value problem
\[
\frac{d^2 \psi}{dx^2} + \frac{2mE}{\hbar^2} \psi = 0 \quad ; \quad \psi(0) = \psi(\ell) = 0
\] (11.59)

Especially if we substitute the shorthand \( k^2 = 2mE/\hbar^2 \) (or \( E = \hbar^2 k^2 / 2m \)), we recognize that this quantum mechanical problem is mathematically identical to the classical standing wave problem described in Section 11.1.8. This problem admits solutions only for a discrete set of special values for \( k \), i.e., only for special energies.

Suppose, alternatively, that we take the potential energy to be \( V(x) = \frac{1}{2}kx^2 \), which describes the quantum analog to the classical harmonic oscillator with spring constant \( k \). In this case, the time-independent Schrödinger equation becomes
\[
\frac{d^2 \psi}{dx^2} + \frac{2m}{\hbar^2} \left( E - \frac{1}{2}kx^2 \right) \psi = 0
\] (11.60)

or, in dimensionless form,
\[
\frac{d^2 \psi}{d\tau^2} + (2\epsilon - \tau^2) \psi = 0
\] (11.61)

where \( \omega = \sqrt{k/m} \), \( \epsilon = E/\hbar \omega \) and \( \tau = x \sqrt{\hbar/m\omega} \). In contrast to the infinitely deep well, the domain for the quantum oscillator extends over the interval \( -\infty < x < +\infty \), and we must require that \( \psi(x) \) approach zero as \( x \) approaches either end of this interval. Numerically, infinite domains
are complicated. Note, however, that the equation in this case is quadratic in \( x \). Thus, the solutions to the equation can be divided into two sets, one of which contains functions that are even in \( x \) (even parity) and the other of which contains functions that are odd in \( x \) (odd parity). Even functions, however, necessarily have zero derivative and non-zero value at \( x = 0 \) while odd functions, in contrast, have non-zero derivative and zero value at \( x = 0 \). These properties mean that we can replace the original boundary value problem involving Eq. (11.60) over an infinite interval with an initial value problem for the solution over half of the infinite interval. When we come later to address the actual solution of this problem, we will therefore focus on two sub-problems, one defined by the equation and initial values

\[
\frac{d^2\psi}{dx^2} + (2\epsilon - x^2)\psi = 0 \quad ; \quad \psi(0) = 1.0 \quad ; \quad \frac{d\psi}{dx}(0) = 0 \quad ; \quad \lim_{x \to \infty} \psi(x) = 0
\] (11.62)

and yielding even solutions and the other defined by the equation and initial values

\[
\frac{d^2\psi}{dx^2} + (2\epsilon - x^2)\psi = 0 \quad ; \quad \psi(0) = 0 \quad ; \quad \frac{d\psi}{dx}(0) = 1 \quad ; \quad \lim_{x \to \infty} \psi(x) = 0
\] (11.63)

and yielding odd solutions. The interesting aspect of these problems is that we will be able to find solutions satisfying the boundary requirement at \( x = +\infty \) only for very special values of \( \epsilon \) in the equation, i.e., only for special values of the energy.

### 11.2 Laplace Transforms

One tool used behind the scenes by symbolic solvers of ODEs is called the Laplace transform, which we defined and illustrated in Section 1.5.2. We return to the topic here, not because we are likely to make much use of it directly but because knowing its properties and capabilities may sometimes be valuable as we try to guide a symbolic manipulator that uses the technique. As we laid out in the referenced section, the Laplace transform is defined for a function \( f(t) \) by the integral

\[
\mathcal{L}\left(f(t)\right) = \tilde{f}(s) = \int_0^\infty e^{-st} f(t) \, dt
\] (11.64)

We can best illustrate the value of the Laplace transform in solving a linear, ordinary differential equation with constant coefficients by presenting a short example. We would—and symbolic manipulators may well—apply this technique to the third member of Eq. (11.4) as follows. First we use Eqs. (1.16) and (1.17) and entries in Table 1.2 to evaluate its Laplace transform, finding that

\[
m[s^2\tilde{z}(s) - sz_0 - v_{z0}] = -\frac{mg}{s} - b[s\tilde{z}(s) - z_0]
\] (11.65)

Then we solve this algebraic equation\(^\text{12}\) for \( \tilde{z}(s) \), finding that

\[
\tilde{z}(s) = \frac{z_0}{s} + \frac{v_{z0}s - g}{s^2(s + b/m)}
\] (11.66)

Finally, we invert this Laplace transform to find the solution itself. Unfortunately, inverting Laplace transforms is rarely easy.\(^\text{13}\) We can, of course, read a table of transforms such as Table 1.2 backwards,

---

\(^\text{10}\)It involves only second derivatives and the potential energy involves only \( x^2 \).

\(^\text{11}\)Reducing infinity to half of infinity may not seem like much of a simplification. The primary value of the change is that the starting point for tracking a solution has now been moved into an accessible region.

\(^\text{12}\)The immense simplification coming from using the Laplace transform is precisely the conversion of a differential equation (of the right sort) into an algebraic equation for the transform of the solution.

\(^\text{13}\)Murphy's law, in the version that says that transforms may move difficulties around but may not eliminate them altogether, applies. The price we pay for the conversion to an algebraic equation is that we now have to invert the transform we found so easily.
but that strategy will work only if the entry we need is in fact included in the table. In the present case, however, we can recast the result we have obtained by using the technique of partial fractions to find the equivalent expression

\[ \hat{z}(s) = \frac{z_0}{s} - \frac{mg/b}{s^2} + \frac{m}{b} \left(v_{z0} + \frac{mg}{b}\right) \left(\frac{1}{s} - \frac{1}{s + b/m}\right) \]  \hspace{1cm} (11.67)

All of the pieces in this form of the transform now are in even the tiny table we have available. Since the inverse of a sum of terms is the sum of the inverses of each term separately, we conclude that

\[ z(t) = z_0 - \frac{mg}{b} \cdot t + \frac{m}{b} \left(v_{z0} + \frac{mg}{b}\right) \left(1 - e^{-bt/m}\right) \]  \hspace{1cm} (11.68)

Note that, in this approach, the initial conditions imposed on the solution are incorporated ab initio, not imposed after a general solution with arbitrary, unknown constants has been obtained.

### 11.6 Algorithms for Solving ODEs Numerically

A numerical solution to an ordinary differential equation emerges from the application of a procedure—frequently called an algorithm—for calculating approximate values of the dependent variables at a succession of values of the independent variable. All numerical methods for solving ordinary differential equations exploit the fact that the differential equations determine the rates of change of the dependent variables from the dependent variables themselves. Because solutions obtained numerically are approximate, we must give attention not only to the methods themselves but also to means by which we can assess the accuracy of the solutions obtained.

For the sake of a simple discussion, we shall, in laying out the essence of each of several algorithms, suppose that we are dealing with a single first-order ODE and an initial condition of the form

\[ \frac{dx}{dt} = f(x, t) \hspace{0.5cm} ; \hspace{0.5cm} x(0) = x_0 \]  \hspace{1cm} (11.69)

where \( f(x, t) \) and \( x_0 \) are known from the beginning. The relatively straightforward extension of the initial discussion to systems of first-order equations and to single equations of higher order will be illustrated in several specific examples but will not be explicitly discussed in general terms.

#### 11.6.1 Euler’s Method

Euler’s method, which embodies the simplest numerical approach to ODEs, is based on the assumption that the rates of change of the dependent variables do not themselves change very quickly. Thus, given a short enough time interval \( \Delta t \), the rates of change throughout that interval may be regarded, at least approximately, as constant and equal to the rates of change at the beginning of the interval. For example, provided \( \Delta t \) is not too large,\(^{14}\) we can write the approximation

\[ \frac{dx}{dt} = f(x, t) \implies \frac{x(t + \Delta t) - x(t)}{\Delta t} \approx f(x, t) \implies x(t + \Delta t) \approx x(t) + f(x, t) \Delta t \]  \hspace{1cm} (11.70)

The value \( x(t + \Delta t) \) of the dependent variable at \( t + \Delta t \) is the value \( x(t) \) at time \( t \) plus the amount \( \Delta x \) by which \( x \) changes in the interval, where we estimate \( \Delta x \) by multiplying the rate of change of \( x \) given by \( f(x(t), t) \) at the beginning of the interval by the elapsed time \( \Delta t \), i.e., \( \Delta x = f(x(t), t) \Delta t \).

In an alternative notation, if we think of \( t \) as the “old” time and \( t + \Delta t \) as the “new” time, we might express the basic stepping equations as

\[ f_{\text{old}} = f(x_{\text{old}}, t_{\text{old}}) \hspace{0.5cm} ; \hspace{0.5cm} x_{\text{new}} = x_{\text{old}} + f_{\text{old}} \Delta t \hspace{0.5cm} ; \hspace{0.5cm} t_{\text{new}} = t_{\text{old}} + \Delta t \]  \hspace{1cm} (11.71)

\(^{14}\)The precise meaning of “too large” is difficult to define. In general terms, if \( T \) is a typical time during which the solution changes appreciably (say by 10–20%), then a value of \( \Delta t \) satisfying \( \Delta t \ll T \) will probably yield an adequate solution. Each case must be examined on its own terms; no general rules can be formulated.
### 11.6. ALGORITHMS FOR SOLVING ODES NUMERICALLY

Table 11.1: Simple Euler algorithm.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dt ← ∆t</code></td>
<td>Choose time step.</td>
</tr>
<tr>
<td><code>told ← 0.0</code></td>
<td>Initialize <code>t_{old}</code>.</td>
</tr>
<tr>
<td><code>xold ← x_0</code></td>
<td>Initialize <code>x_{old}</code>.</td>
</tr>
<tr>
<td><code>loop</code></td>
<td></td>
</tr>
<tr>
<td><code>fold ← f(xold, told)</code></td>
<td>Evaluate <code>f_{old}</code>.</td>
</tr>
<tr>
<td><code>print, told, xold, fold</code></td>
<td>Display results.</td>
</tr>
<tr>
<td><code>exit when done</code></td>
<td></td>
</tr>
<tr>
<td><code>xnew ← xold + fold*dt</code></td>
<td>Calculate values at new time.</td>
</tr>
<tr>
<td><code>tnew ← told + dt</code></td>
<td></td>
</tr>
<tr>
<td><code>xold ← xnew</code></td>
<td>Replace old values with new.</td>
</tr>
<tr>
<td><code>told ← tnew</code></td>
<td></td>
</tr>
<tr>
<td><code>end_loop</code></td>
<td></td>
</tr>
</tbody>
</table>

Starting from the initial condition as the first `x_{old}` (and a choice of time step `∆t`—see later), we can then use these stepping equations repeatedly to move from knowledge of `x(0)` to knowledge of `x(∆t)` to knowledge of `x(2∆t)` to ..., continuing as long as our patience endures. The basic strategy of Euler's method would be summarized in the algorithm in Table 11.1.\(^{15}\) As this example shows, we can generate a complete—though approximate—solution from initial knowledge of only

1. The differential equation, which determines the rate of change of the dependent variable from the dependent and independent variables themselves,
2. The initial condition, which starts the process by providing the first row in a table containing `t`, `x`, and `f`,
3. Specific values for any parameters—here there happen to be none—in the differential equation, and
4. A choice of time step `∆t`.

With this information as input, each pass through the loop in the above algorithm generates a new row in the table containing `t`, `x`, and `f`.

To be even more concrete (and to illustrate the simple extension to a system of ODEs), let us work out by hand the first few steps in the Euler solution of Eq. (11.19) describing a chain radioactive decay. Reflecting the differential equations, the stepping equations for this specific case are

\[
A(t + ∆t) = A(t) + \frac{dA}{dt}(t) ∆t = A(t) - k_A A(t) ∆t \quad (11.72)
\]

\[
B(t + ∆t) = B(t) + [k_A A(t) - k_B B(t)] ∆t \quad (11.73)
\]

\[
C(t + ∆t) = C(t) + k_B B(t) ∆t \quad (11.74)
\]

For definiteness,\(^{16}\) we take `A(0) = 1000.000`, `B(0) = C(0) = 0.000`, and `k_A = k_B = 0.100`, and we

---

\(^{15}\)In many cases, the ultimate and penultimate pairs of statements in the loop can be combined into a single pair of statements. We here refrain from that more compact expression so that the two distinct operations—calculating the new to complete the step and replacing the old with the new to prepare for the next step—will remain distinct.

\(^{16}\)One disadvantage of numerical approaches is that they are, indeed, numerical. We cannot find solutions containing symbols representing parameters. We must seek solutions for specific numerical values. If we need to know the dependence of the solution on a particular parameter, we will have to generate a separate solution for each desired value of the parameter.
Table 11.2: Evolution of three-species radioactive decay determined by Euler’s method with $\Delta t = 0.25$.

<table>
<thead>
<tr>
<th>$t$</th>
<th>$A$</th>
<th>$B$</th>
<th>$C$</th>
<th>$A + B + C$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1000.000</td>
<td>0.000</td>
<td>0.000</td>
<td>1000.000</td>
</tr>
<tr>
<td>0.25</td>
<td>975.000</td>
<td>25.000</td>
<td>0.000</td>
<td>1000.000</td>
</tr>
<tr>
<td>0.50</td>
<td>950.625</td>
<td>48.750</td>
<td>0.625</td>
<td>1000.000</td>
</tr>
<tr>
<td>0.75</td>
<td>926.859</td>
<td>71.297</td>
<td>1.844</td>
<td>1000.000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

select a time step of $\Delta t = 0.250$. Equations (11.72), (11.73), and (11.74) at $t = 0$ then become

\[
\begin{align*}
A(0.00 + 0.25) &= 1000.0 - 0.1 \cdot 1000.0 \cdot 0.25 = 975.000 = A(0.25) \\
B(0.00 + 0.25) &= 0.0 + [0.1 \cdot 1000.0 - 0.1 \cdot 0.0] \cdot 0.25 = 25.000 = B(0.25) \quad (11.75) \\
C(0.00 + 0.25) &= 0.0 + 0.1 \cdot 0.0 \cdot 0.25 = 0.000 = C(0.25)
\end{align*}
\]

Continuing the algorithm further by applying Eqs. (11.72), (11.73), and (11.74) for $t = 0.25$, we find that

\[
\begin{align*}
A(0.25 + 0.25) &= 975.0 - 0.1 \cdot 975.0 \cdot 0.25 = 950.625 = A(0.50) \\
B(0.25 + 0.25) &= 25.0 + [0.1 \cdot 975.0 - 0.1 \cdot 25.0] \cdot 0.25 = 48.750 = B(0.50) \quad (11.76) \\
C(0.25 + 0.25) &= 0.0 + 0.1 \cdot 25.0 \cdot 0.25 = 0.625 = C(0.50)
\end{align*}
\]

Going yet one more step, Eqs. (11.72), (11.73), and (11.74) for $t = 0.50$ yield that

\[
\begin{align*}
A(0.50 + 0.25) &= 950.625 - 0.1 \cdot 950.625 \cdot 0.25 = 926.859 = A(0.75) \\
B(0.50 + 0.25) &= 48.75 + [0.1 \cdot 950.625 - 0.1 \cdot 48.75] \cdot 0.25 = 71.267 = B(0.75) \quad (11.77) \\
C(0.50 + 0.25) &= 0.625 + 0.1 \cdot 48.75 \cdot 0.25 = 1.844 = C(0.75)
\end{align*}
\]

The resulting values are compiled in Table 11.2, in which the first row is provided by the initial conditions and all subsequent rows are determined by the algorithm outlined in Eqs. (11.72), (11.73), and (11.74). Knowing that an exact solution will reflect the conservation of $A + B + C$ [see Eq. (11.20)], we have added a column containing the sum of the amounts of all three species present. Clearly, carried this far, anyway, the solution appears automatically to satisfy that requirement.\(^{18}\)

### 11.6.2 Improved Euler Method

While Euler’s method is simple to motivate, describe, and implement, it unfortunately yields only a coarse approximation. Typically it will require a *very* small time step (which translates into a large amount of computational time and the potential accumulation of computer roundoff error) to achieve adequate accuracy. Considerable effort has been spent in devising alternative, more refined algorithms that converge more rapidly with a minimal amount of computational labor (and hence less internal roundoff error). The *improved Euler method*, for example, takes the Euler result at each step to be only an *estimate* (prediction) of the solution for that step and uses that estimate to refine the solution before going on to the next step. Starting with the values $x_{old}$ and $t_{old}$ and a chosen time step $\Delta t$, we

\(^{17}\)This choice at the moment is more for convenience than accuracy. The accuracy of the resulting solution is, of course, markedly influenced by the choice of $\Delta t$. We shall return to assess the suitability of this choice in later sections.

\(^{18}\)Actually, however, that the sum $A + B + C$ preserves its initial value is not really a check on the accuracy of the method in *this example*. If we simply add the three stepping equations we are using [Eqs. (11.72)–(11.74)], we find that $A(t + \Delta t) + B(t + \Delta t) + C(t + \Delta t) = A(t) + B(t) + C(t)$. *Solutions generated by Euler’s method applied to this problem automatically satisfy the conservation law, regardless of $\Delta t$.*
Table 11.3: Improved Euler algorithm.

```
dt ← ∆t

told ← 0.0

xold ← x0

loop

fold ← f(xold, told)
print, told, xold, fold
exit_loop when done

xpred ← xold + fold*dt

tnew ← told + dt

fpred ← f(xpred, tnew)

xnew ← xold + 1/2(fold + fpred)*dt

xold ← xnew

told ← tnew

end_loop
```

Choose time step.

Initialize \( t_{old} \).

Initialize \( x_{old} \).

Evaluate \( f_{old} \).

Display results.

Calculate predicted value.

Evaluate \( f_{pred} \).

Calculate final (corrected) value.

Replace old values with new.

1. calculate \( f_{old} \),

2. calculate the predicted values

\[
x_{pred} = x_{old} + f_{old} \Delta t ; \quad t_{new} = t_{old} + \Delta t
\]  \hspace{1cm} (11.78)

3. calculate \( f_{pred} = f(x_{pred}, t_{new}) \), and

4. calculate the final (corrected) value

\[
x_{new} = x_{old} + \frac{1}{2} (f_{old} + f_{pred}) \Delta t
\]  \hspace{1cm} (11.79)

In this last step, we in effect estimate the average rate of change over the interval from \( t \) to \( t + \Delta t \) as the average of (a) its value \( f_{old} \) at the beginning of the interval and (b) the best estimate \( f_{pred} \) we have of its value at the end of the interval. In so doing, we admit that the rate of change may change in the interval. Intuitively, for a given \( \Delta t \), the improved Euler method will be more accurate than Euler’s method, which presumes that the average rate of change over the interval is adequately approximated by its value at the beginning of the interval.\(^1\)

A full laying out of this improved algorithm differs from the algorithm presented in the previous section in only a few lines, having the expression laid out in Table 11.3. As with Euler’s method, knowledge of the differential equation, the initial condition, and any parameters, together with a choice of a time step, starts a process that leads to (approximate) knowledge of the solution indefinitely into the future.

By way of example (but leaving the arithmetic to the reader), we present in Table 11.4 the results of applying the improved Euler method for the example treated in the previous section. Values in this table should be compared with those in Table 11.2.

### 11.6.3 Runge-Kutta Methods

A popular alternative viewpoint, which leads to the deduction of what are called Runge-Kutta stepping algorithms, requires at base that two different Taylor series expansions of the solution

\(^1\)Phrased as we have described it, the improved Euler method entails first calculating a predicted value at the end point of an interval and then uses that value to determine a corrected final value at that point before going on to the next point. Such methods are often called predictor-corrector methods.
Chapter 11. Solving Ordinary Differential Equations

Table 11.4: Evolution of three-species radioactive decay determined by the improved Euler method with $\Delta t = 0.25$.

<table>
<thead>
<tr>
<th>t</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>A + B + C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1000.00</td>
<td>0.000</td>
<td>0.000</td>
<td>1000.000</td>
</tr>
<tr>
<td>0.25</td>
<td>975.313</td>
<td>24.375</td>
<td>0.312</td>
<td>1000.000</td>
</tr>
<tr>
<td>0.50</td>
<td>951.234</td>
<td>47.547</td>
<td>1.219</td>
<td>1000.000</td>
</tr>
<tr>
<td>0.75</td>
<td>927.751</td>
<td>69.559</td>
<td>2.690</td>
<td>1000.000</td>
</tr>
</tbody>
</table>

... ... ... ...

match to a chosen number of terms. On the one hand, we know that

$$
x(t + \Delta t) = x(t) + \frac{dx(t)}{dt} \Delta t + \frac{1}{2} \frac{d^2x(t)}{dt^2} \Delta t^2 + O(\Delta t^3)
$$

$$
= x(t) + f(x, t) \Delta t + \frac{1}{2} \left( \frac{\partial f(x, t)}{\partial x} \frac{dx(t)}{dt} + \frac{\partial f(x, t)}{\partial t} \right) \Delta t^2 + O(\Delta t^3)
$$

$$
= x(t) + f(x, t) \Delta t + \frac{1}{2} \left( f(x, t) \frac{\partial f(x, t)}{\partial x} b \Delta t f(x, t) + \frac{\partial f(x, t)}{\partial t} a \Delta t + O(\Delta t^2) \right) \Delta t^2 + O(\Delta t^3)
$$

Motivated by the improved Euler method on the other hand, we are tempted to suppose that we might build a stepping algorithm by

1. introducing a judicious time $t' = t + a \Delta t$, where $a$, constrained by $0 \leq a \leq 1$, is yet to be specified,
2. introducing a tentative solution $x' = x(t) + b \Delta tf(x(t), t)$, where $b$, constrained by $0 \leq b \leq 1$, is yet to be specified, and
3. taking $x(t + \Delta t) = x(t) + \left( w_1 f(x(t), t) + w_2 f(x', t') \right) \Delta t$, where the weights $w_1$ and $w_2$ are yet to be specified.

The essential idea of the Runge-Kutta approach is to expand the assumption of item 3 as a power series in $\Delta t$ and require that its first few terms agree with those in the series expressed in Eq. (11.80). To deduce this second Taylor series, we first invoke the two-dimensional Taylor series to find that

$$
f(x', t') = f \left( x + b \Delta t f(x, t), t + a \Delta t \right)
$$

$$
= f(x, t) + \frac{\partial f(x, t)}{\partial x} b \Delta t f(x, t) + \frac{\partial f(x, t)}{\partial t} a \Delta t + O(\Delta t^2)
$$

where—noting at item 3 above that this term will ultimately be multiplied by $\Delta t$—we have included only terms through those first order in $\Delta t$. Then, we find that

$$
x(t + \Delta t) = x(t) + w_1 f(x, t) \Delta t
$$

$$
+ w_2 \left( f(x, t) + \frac{\partial f(x, t)}{\partial x} b \Delta t f(x, t) + \frac{\partial f(x, t)}{\partial t} a \Delta t + O(\Delta t^2) \right) \Delta t
$$

Note that the steps laid out here reduce to the improved Euler method if we take $a = b = 1$ and $w_1 = w_2 = 1/2$. 
11.6. ALGORITHMS FOR SOLVING ODES NUMERICALLY

Table 11.5: A Runge-Kutta Method.

- \( dt \leftarrow \Delta t \)
- \( t_{old} \leftarrow 0.0 \)
- \( x_{old} \leftarrow x_0 \)

**loop**
- \( fold \leftarrow f(x_{old}, t_{old}) \)
- **print, t_{old}, x_{old}, fold**
- **exit loop when done**
- \( k_1 \leftarrow fold \cdot dt \)
- \( k_2 \leftarrow f(x_{old}+k_1, t_{old}+dt) \cdot dt \)
- \( t_{new} \leftarrow t_{old} + dt \)
- \( x_{new} \leftarrow x_{old} + (k_1+k_2)/2 \)
- \( x_{old} \leftarrow x_{new} \)
- \( t_{old} \leftarrow t_{new} \)

**end loop**

\[
\frac{dx}{dt} = x(t) + (w_1 + w_2)f(x, t) \Delta t \\
+ \left( w_2 b f(x, t) \frac{\partial f(x, t)}{\partial x} + w_2 a \frac{\partial f(x, t)}{\partial t} \right) \Delta t^2 + O(\Delta t^3) \tag{11.82}
\]

which will match Eq. (11.80) through terms of order \( O(\Delta t^2) \) if we choose

\[
w_1 + w_2 = 1 \quad ; \quad aw_2 = bw_2 = \frac{1}{2} \quad \text{or} \quad w_1 + w_2 = 1 \quad ; \quad a = b \quad ; \quad w_2 = \frac{1}{2a} \tag{11.83}
\]

We have discovered a multitude of Runge-Kutta schemes, one corresponding to each possible choice of \( a, b, w_1, \) and \( w_2 \).\(^{21}\) One common choice embodies the values in footnote 20—values that reveal that the improved Euler method is a member of this Runge-Kutta family. A second common choice takes \( w_1 = 0, w_2 = 1 \), and \( a = b = \frac{1}{2} \), in which case the stepping equations become

\[
t' = t + \frac{1}{2} \Delta t \quad ; \quad x' = x + \frac{1}{2} f(x, t) \Delta t \quad ; \quad x(t + \Delta t) = x(t) + f(x', t') \Delta t \tag{11.84}
\]

With *this* scheme, we first step half way over the interval with Euler’s method, then we use that result to estimate the rate of change at the midpoint of the interval and, finally, we use the rate of change at the midpoint to project the solution at the end of the interval. Clearly, this route provides yet another means to estimate the average rate of change over the interval; it is called the **midpoint method**.

Even though the Runge-Kutta algorithm for \( w_1 = w_2 = \frac{1}{2} \) and \( a = b = 1 \) coincides with the improved Euler method, it is usually presented in the form shown in Table 11.5. This form is more compatible with the most convenient expressions of other algorithms in the broad Runge-Kutta family.

The algorithm described in the previous two paragraphs is known as a **second-order** Runge-Kutta algorithm because its deduction entailed matching terms in two Taylor expansions through those of order \( \Delta t^2 \). We could, of course, replace the expression in item 3 above with a more elaborate expression, expand it to include higher-order terms in \( \Delta t \), and insist on agreement with a higher-order Taylor expansion deduced from the differential equation. The calculational labor becomes increasingly complicated. Partly because of its popularity, we present without derivation the essence of a **fourth-order** Runge-Kutta algorithm, limiting ourselves only to the steps that would replace the ones calculating \( k_1, k_2, \) and \( x_{new} \) in the above algorithm:

\(^{21}\) Actually, we choose only \( a \). Then \( b = a, w_2 = 1/2a, \) and \( w_1 = 1 - w_2 \) are fixed.
k1 ← f(xold, told)*dt  
Find change using rate at beginning.

k2 ← f(xold+k1/2, told+dt/2)*dt  
Estimate change using rate at midpoint.

k3 ← f(xold+k2/2, told+dt/2)*dt  
Estimate change using refined rate at midpoint.

k4 ← f(xold+k3, told+dt)*dt  
Estimate change using rate at end.

xnew ← xold + (k1+2*k2+2*k3+k4)/6  
Calculate final value.

This fourth-order algorithm emerges when the two Taylor series are matched through the terms involving $\Delta t^4$. A higher-order method, of course, is more accurate than a lower-order method for a given time step. Equivalently, a lower-order method requires a smaller time step to give the same degree of accuracy as a higher-order method.

### 11.6.4 Assessing Accuracy

Numerical evaluations, of course, only approximate the solution to ordinary differential equations. Thus, we cannot complete a solution without also assessing its accuracy. Furthermore, this task must be accomplished without knowledge of the exact solution. The importance of being aware that numerical methods are always approximate cannot be overstressed.

Two distinctly different sorts of errors can occur. *Truncation errors* arise because the solution has been based on a finite-difference approximation to the derivatives appearing in the equation; *roundoff errors* arise because computers do not store non-integers to 100% precision and, in an iterated calculation where each step depends on the previous step, the imprecision with which each component is represented within the computer can accumulate as the number of steps increases. Truncation errors become smaller as the step size is reduced. Roundoff errors, unfortunately, become more significant as step size is reduced (because, with smaller steps, more arithmetic must be done). Usually, roundoff errors are negligible, the more so as the sophistication of the algorithm increases (and, hence, the amount of arithmetic decreases). Provided we do not strive for accuracy greater than about 1 part in $10^5$ (with single precision floating point arithmetic), we can usually ignore roundoff errors. Thus, provided the solution we seek does not vary too rapidly on the time scale defined by the time step in use, the quickest way to obtain a reasonably reliable estimate of truncation error is to solve the equation with two different step sizes, the second being half of the first, and compare the two results. Presuming that roundoff error has not begun to be important, we can be confident that the second result is more accurate than the first. Thus, if the two agree to 1 part in $10^3$, say, we can with reasonable confidence assume that the second value is good to one part in $10^3$. Indeed, the second value is probably better than that, but assessing its accuracy by this method would entail obtaining a third value by using a step size half of that used to determine the second value. Indeed, one strategy for achieving a desired accuracy with reasonable certainty is to solve the ODE repeatedly by a particular method, halving the step size each time, and continuing until the new value received differs from its predecessor by less than the desired accuracy (though we must be careful not to push this approach so far that roundoff problems within the computer begin to become significant). We will illustrate this approach in the context of a specific example as soon as we are ready to use the computer to do the arithmetic.

From a more sophisticated perspective, numerical analysts have deduced expressions for the error in various approaches to solving ODEs numerically. To assess the error in Euler’s method, for example, we begin by noting the Taylor theorem with remainder, which asserts that

$$
x(t + \Delta t) = x(t) + \frac{dx}{dt}(t) \Delta t + \frac{1}{2} \frac{d^2x}{dt^2}(\xi) \Delta t^2
$$

(11.85)

where $\xi$ is a value in the interval $t \leq \xi \leq t + \Delta t$. This expression is exact, though it is only somewhat useful because it tells us only the order of magnitude of the error; it gives no clue as to the actual

---

22 which provides yet another motivation for casting problems in dimensionless form.

23 This assertion is refined in the next paragraphs.

24 **Warning**: This approach is not entirely secure if the procedure converges slowly as the time step is reduced. Remember that $\sum_{n=1}^{\infty} \frac{1}{n}$ diverges even though the effect of the millionth term added to the partial sum starts in the sixth decimal place.
value of $\xi$. Nonetheless, we can conclude that

$$x_{\text{exact}}(t + \Delta t) = x_{\text{Euler}}(t + \Delta t) + \frac{1}{2} \frac{d^2 x}{dt^2}(\xi) \Delta t^2$$  \hspace{1cm} (11.86)

or that

$$\left| x_{\text{exact}}(t + \Delta t) - x_{\text{Euler}}(t + \Delta t) \right| \leq \text{Max} \left( \left| \frac{1}{2} \frac{d^2 x}{dt^2}(\xi) \right| \right) \Delta t^2$$  \hspace{1cm} (11.87)

Thus, we learn that, with Euler’s method, the (truncation) error per step varies as the square of the step size; halving the step size will reduce the error per step by a factor of four.

Statements similar to Eq. (11.86) can be deduced for all of the methods that we have described in this section. Without proof,25 we present the properties

$$x_{\text{exact}}(t + \Delta t) = x_{\text{Euler}}(t + \Delta t) + O(\Delta t^2)$$  \hspace{1cm} (11.88)
$$x_{\text{ImpEul}}(t + \Delta t) + O(\Delta t^3)$$  \hspace{1cm} (11.89)
$$x_{\text{RK2}}(t + \Delta t) + O(\Delta t^3)$$  \hspace{1cm} (11.90)
$$x_{\text{RK4}}(t + \Delta t) + O(\Delta t^5)$$  \hspace{1cm} (11.91)

For these four common methods, halving the step size reduces the error per step by a factor of four, eight, eight, and thirty-two, respectively. Further, on the basis of these relationships, we characterize “Euler” as a first-order method, “ImpEul” and “RK2” as second-order methods, and “RK4” as a fourth-order method because their derivations involve matching Taylor series to include terms in $\Delta t$, $\Delta t^2$, $\Delta t^3$, and $\Delta t^4$, respectively.

Unfortunately, the analysis described briefly in the previous paragraph is not the whole story. Each of the statements in that paragraph is correct, provided we assume that the step that arrives at the various estimates of $x(t + \Delta t)$ starts with the exact solution at time $t$. Except for the first step, which moves away from (exact) initial conditions, that assumption is invalid. Truncation errors per step compound as more and more steps are taken, and a full assessment of the error in a solution obtained by one or another numerical means must recognize this cumulation. Beyond the error per step, we must be aware of the global truncation error, which attempts to estimate how much error accumulates in the course of working out a solution over the entire desired range of the independent variable. The task of assessing global error is extremely difficult. Crudely, however, if we simply add up expressions like those in the previous paragraph for the $N$ steps in an entire solution and suppose—without much justification—that each step contributes about the same amount, we would conclude that the global truncation error would be order $N O(\Delta t^p)$ when the error per step is of order $O(\Delta t^p)$. For a fixed interval, $N$ is itself of order $1/\Delta t$. Thus, we infer that the global truncation error is of order $O(\Delta t^{p-1})$. This result does not help us much in determining the global error. It does, however, support the conclusion that, for the four methods in the previous paragraph, halving the step size will reduce the global error by a factor of two, four, and sixteen, respectively. In particular, halving the step size with the fourth-order Runge-Kutta method will add at least one more decimal digit to the accuracy of the solution overall (and may well do much better).

We have already mentioned that conserved quantities like energy, linear momentum, and angular momentum can sometimes also be used as a check on the accuracy of an evolving solution. Whenever such a conserved quantity exists, its initial value must, of course, be preserved (within some limits). Failure of a particular solution to conform to that requirement signals a need for a smaller time step or a more sophisticated algorithm. Note, however, that some algorithms (e.g., the Euler algorithm for the three-species radioactive decay) automatically preserve one or more conserved quantities; preservation in such cases does not provide any information about the accuracy of the solution itself.

---

11.6.5 Adaptive Methods

In the previous subsections, we assumed that the user of a particular algorithm would actually view the values obtained for different time steps and decide personally when to stop by examining the changes that occur as the time step is successively halved. One can, of course, program a computer to make those decisions. In essence, the program generates the solution with one time step, then repeatedly generates it with a succession of ever smaller time steps, comparing each new solution with its immediate predecessor and stopping when the absolute value of the difference is smaller than a tolerance—either absolute or relative—prescribed in advance. As a guard against an infinite loop, these algorithms should also stop if the desired tolerance has not been achieved in some maximum number of refinements and should print a warning when the desired tolerance has not in its judgment been achieved.

Another family of algorithms aims to minimize computational labor by estimating—though the methods for doing so are often themselves approximate—the accuracy obtained at each step along the way to a solution. A trial row in the table of data is generated, and the error is assessed. If the error is within a user-specified tolerance, the program moves on to the next row in the data table. If the error exceeds the desired tolerance, the program repeats the calculation with progressively smaller time steps until the desired tolerance is achieved and only then is a new row added to the table. The procedure also contains means by which the time step is increased when the estimated error is less than the specified tolerance. Thus, the time step fluctuates as the solution unfolds, being small when the solution is changing rapidly and large when the solution is changing slowly. Because of this feature, the procedure is said to be adaptive. Without the overlay of an elaborate interpolation, adaptive methods have the disadvantage of generating solutions at irregularly spaced times. That disadvantage, however, is frequently outweighed by the substantial advantage of concentrating the computational effort in regions where the solution changes rapidly.

11.6.6 Multistep Methods

All of the methods so far discussed have generated the solution at a particular point from knowledge of the solution at a single earlier point, though many have interpolated solutions at several points between the initially known point and the desired end point; such methods are called single-step methods. Some of the solvers available in some software packages use multistep methods, which reduce the need for the interpolative procedures by projecting the solution at the next point from knowledge of the solution at several previous points. In seeking a solution to Eq. (11.69), for example, we would choose a time step $\Delta t$ and introduce the points $t_i = i \Delta t$ and the notation $x_i = x(t_i)$ and $f_i = f(x_i, t_i)$. Then, supposing we already had in hand estimates of the solution at four (say) earlier points, we might estimate the solution at $t_{i+1}$ by the formula

$$x_{i+1} = x_i + \frac{\Delta t}{24} \left( 55f_i - 59f_{i-1} + 37f_{i-2} - 9f_{i-3} \right)$$

(11.92)

(which represents a particular fourth-order member of a family of methods known as Adams-Bashforth methods). In effect, we are estimating the average derivative over the interval $t_{i+1} \leq t \leq t_i$ as a weighted average of the derivatives we can compute at the four points previous to the one at $t_{i+1}$, all of which we would—of course—have to have in hand at the time we took this step.

Because they do not require interpolation within the interval over which we are stepping at any moment, multistep methods can be computationally extremely efficient. Since we cannot use such a

---

26 Sometimes the assessment of accuracy will be based not on the generated solution itself but on some property, e.g., the energy, computed from that solution.

27 In contrast, Euler’s method, for example, would require using throughout a time step small enough to generate an accurate solution in regions where the solution varies most rapidly. Consequently, too much computational effort would be focused in regions where the solution changes slowly enough that a larger time step would be acceptable.

method until we have in hand the solution at several—in the above example four—points, however, these methods are not self starting. We must adopt some other method to obtain from the initial values however many values are needed to support the multistep method. To start a solution based on the stepping formula of Eq. (11.92), we might use a fourth-order Runge-Kutta method to obtain \( x_1, x_2, \) and \( x_3 \) from \( x_0 \). Then we could shift to the fourth-order Adams-Bashforth formula for all subsequent points.

We could, of course, simply take the result given by Eq. (11.92) as the solution at \( t_{i+1} \). For a more sophisticated solution, we could regard that result as a prediction and use it in a paired formula to “correct” the prediction. An appropriate fourth-order corrector formula is a member of the Adams-Moulton family, namely

\[
x_{i+1} = x_i + \frac{\Delta t}{24} \left( 9f_{i+1} + 19f_i - 5f_{i-1} + f_{i-2} \right)
\]  

(11.93)

Once a sufficient number of values is in hand, we could at each step invoke Eq. (11.92) to predict the next solution and then Eq. (11.93) to refine it. We might call the resulting method the Adams-Bashforth-Moulton method.

## 11.15 Solving ODEs Numerically with C

C programs to solve ordinary differential equations numerically can be constructed in several ways. In this section, we describe how we can implement one or another algorithm directly in a program that we write from scratch—a task that is tedious and difficult except for the simplest of algorithms. In addition, we describe how we can make use of available standard subroutines—we here focus on those in the _Numerical Recipes_ library—and devote our efforts solely to the easier task of writing a suitable driving program to invoke the features of whatever existing ODE solver we choose to use.

### 11.15.1 C Programs from Scratch

*Note:* All C programs (*.c) and all C-generated data files (*.c.dat) in this subsection can be copied from the directory `$HEAD/cc`. `$HEAD` for your site is defined in the _Local Guide_.

More to illustrate the essence of an algorithm than to develop a truly useful solver, we begin by presenting a quick C program to implement Euler’s method as expressed in Eqs. (11.72)–(11.74) for the three-species radioactive decay. The program contains three major sections, the first of which obtains controlling input (parameters, step size, and initial values) and initializes the time variable, and is expressed with the coding

```c
printf( "\nDecay constant for A: " ); scanf( "%f", &kA );
printf( "Decay constant for B: " ); scanf( "%f", &kB );
printf( "Number of steps : " ); scanf( "%d", &nsteps );
printf( "Time step : " ); scanf( "%f", &dt );
printf( "Initial A : " ); scanf( "%f", &A );
printf( "Initial B : " ); scanf( "%f", &B );
printf( "Initial C : " ); scanf( "%f", &C );
t = 0.0;
```

Then, we open a file to receive the output and write both a labeling first line and the initial values to that file with the coding

```c
```
Table 11.6: Solution loop for decay.c.

```c
for( i=1; i<=nsteps; i++ )
{
    dAdt = -kA*A;  /* Calculate current derivatives */
    dBdt = kA*A - kB*B;
    dCdt = kB*B;
    A = A + dAdt*dt;  /* Calculate new values */
    B = B + dBdt*dt;
    C = C + dCdt*dt;
    t = t + dt;
    fprintf( fptr, "%6.2f %10.3f %10.3f %10.3f\n", t, A, B, C );
}
fclose( fptr );
```

```c
fptr = fopen( "decay_c.dat", "w" );
fprintf( fptr, " t A B C\n" );
fprintf( fptr, "%6.2f %10.3f %10.3f %10.3f\n", t, A, B, C );
```

Finally, we execute a loop in which, for each pass, the derivatives at the current point are evaluated, the variables A, B, C, and t are advanced via the Euler algorithm to values appropriate to the next step, and those values are added to the growing file. The statements completing the calculation, including a closing of the file at the end, are listed in Table 11.6. A full listing of this program, which is named decay.c and includes inclusion of necessary header files and variable declarations, will be found in Section 11.D.

After storing the file containing this program in the file decay.c, we compile and run it to generate the solution with the statements

```bash
cc -o decay.xc decay.c
./decay.xc
```

| Decay constant for A: | 0.1 |
| Decay constant for B: | 0.1 |
| Number of steps       | 200 |
| Time step             | 0.25|
| Initial A             | 1000.0|
| Initial B             | 0.0 |
| Initial C             | 0.0 |

The output is written into the file decay_c.dat, whose first and last few lines are shown in Table 11.7. Reassuringly, the first few lines agree with the results in Table 11.2.

One of the exercises invites you to repeat—and expand—the discussion of this section by recasting decay.c to use the second-order Runge-Kutta algorithm.

---

30To be specific, both here and throughout this section, we illustrate these steps with the statements that would be used in UNIX to compile, link, and run the program. Other operating systems probably accomplish the same end with different statements. In particular, preceding the program name with ./ is necessary in UNIX but may not be necessary with other operating systems. Details will be found in the Local Guide.
Table 11.7: The first and last few lines in the file decay.c.dat.

<table>
<thead>
<tr>
<th>t</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1000.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>0.25</td>
<td>975.000</td>
<td>25.000</td>
<td>0.000</td>
</tr>
<tr>
<td>0.50</td>
<td>950.625</td>
<td>48.750</td>
<td>0.625</td>
</tr>
<tr>
<td>0.75</td>
<td>926.859</td>
<td>71.297</td>
<td>1.844</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>49.50</td>
<td>6.651</td>
<td>33.769</td>
<td>959.580</td>
</tr>
<tr>
<td>49.75</td>
<td>6.485</td>
<td>33.091</td>
<td>960.424</td>
</tr>
<tr>
<td>50.00</td>
<td>6.323</td>
<td>32.426</td>
<td>961.252</td>
</tr>
</tbody>
</table>

11.15.2 Using Numerical Recipes

Note: CPSUP-generated C programs (*.c) and C-generated data files (*.c.dat) can be downloaded from the directory $HEAD/cc. If your site has the appropriate license, files in this subsection from the Numerical Recipes library can be downloaded from a subdirectory of the directory $NRHEAD (see Fig. 10.1). $NRHEAD and $HEAD for your site are defined in the Local Guide.

Numerical Recipes provides many interrelated subroutines for solving ODEs. At the bottom of the Numerical Recipe hierarchy are routines—called algorithm routines—which use one or another basic algorithm to advance the solution to a system of ODEs by one step of specified size. These routines are not adaptive, and they make no attempt to assess the accuracy of their output. In this category, we shall focus on the routine rk4.c, which advances the solution one step by the fourth-order Runge-Kutta method, and the routine rkck.c, which advances the solution one step by the fifth-order Cash-Karp Runge-Kutta method. Other routines in this category, including mmid.c (which implements a modified midpoint method), are used in the same way.

One rung up in the Numerical Recipe hierarchy are routines—called stepper routines—which use an algorithm routine to advance the solution by one step but do so adaptively. Each stepper routine calls an algorithm routine repeatedly with different step sizes until its assessment of the error (per step) satisfies a user-specified tolerance. The stepper routine that uses rkck.c is named rkqs.c; the stepper routine that uses mmid.c is bsstep.c.

We can, of course, use both algorithm and stepper routines directly, and we shall confine ourselves to illustrating those uses. To generate a full solution over some extended interval of the independent variable, therefore, we will have to construct our own loops to execute the routines repeatedly. In doing so, we will be constructing for ourselves versions of programs that constitute the top rung in the Numerical Recipe hierarchy. That is, we shall be constructing routines—called driver routines or, perhaps more accurately in our case, driver programs—which use a stepper algorithm to generate a solution over a given total interval.\(^{31,32}\)

To use an algorithm or stepper routine, we have two responsibilities. First, we must supply a C subroutine that accepts the time (more generally, the independent variable) and a vector containing values of the dependent variables at that time and returns a vector containing the derivatives of the dependent variables at that time. In outline, this file is structured as a subroutine with three arguments, the first—a scalar—providing the current value of the independent variable, the second—a vector—providing the values of the dependent variable, and the third—also a vector—returning the derivatives of the dependent variables. If it should happen that the ODEs to be solved contain

\(^{31}\)Alternatively, we might exploit the driver routines rkdumb.c and odeint.c in the Numerical Recipes library.

\(^{32}\)Remember, too, that the Numerical Recipes library contains numerous example programs illustrating how to use particular routines. As a general rule, the sample programs are in the directory $NRHEAD/recipes_c-ansi/demo/src.\(^{33}\) The sample program corresponding to a particular routine is named by prefixing the character x to the routine name, e.g., xrk4.c for rk4.c.
parameters to which one wishes access outside the subroutine defining the derivatives, we must arrange for the main program to store the parameters in *global variables* (by declaring the variables to be used for the parameters *ahead of* and *outside* all other program blocks and then using the declared variable names both in the function and in the main program when values are assigned to the parameters).\footnote{Details on the use of global variables in C are discussed in Section 9.7.4.} For our standard three-species radioactive decay, for example, we would have to define the two parameters $k_A$ and $k_B$ as global variables with the statement

```c
float kA, kB;    /* For parameters */
```

placed *before* the definition of the subroutine defining the derivatives and *before* the statement `main()` that introduces the main program. Then, we might code the required subroutine—name it `der`—with the statements

```c
void der( float t, float n[], float dndt[] )
{
    dndt[1] = -kA*n[1];    /* Calculate dA/dt */
    dndt[3] = kB*n[2];    /* Calculate dC/dt */
}
```

Second, we must create a driving program that defines appropriate variables, assigns values to all necessary parameters, calls the solver—usually in a loop—to calculate the actual solution, and generates a file containing the results of that calculation. We illustrate first the use of `rk4.c`, which is called with a statement of the form

```c
rk4( ys, dydt, neqs, t, dt, yf, drvs )
```

Here `ys` and `dydt` are vectors containing the values of the dependent variables and their derivatives at time `t`, `neqs` is the number of equations in the system, `dt` is the amount by which the independent variable is to be advanced, `yf` is a vector that receives the values of the dependent variables at time `t+dt`, and `drvs`, which is a string value that is *not* enclosed in quotation marks, is the name of the subroutine that `rk4.c` is to invoke to calculate the derivatives—the subroutine structured as described in the last paragraph.\footnote{In this generic indication of a calling statement, we deliberately use the argument `drvs` rather than `DER` to emphasize that the statement in this paragraph indeed *is* generic. Experience has shown that using `der` in the generic statement, especially right after using it for the name of a specific subroutine, frequently conveys the impression that the only possible name for this argument is `der`—an impression that is decidedly incorrect.} Note (1) that `rk4.c` requires the current values of the derivatives *as input*, so an explicit call to the subroutine returning those derivatives will have to precede the call to `rk4.c`; (2) `yf` can be the *same* vector as `ys`, so the output produced by `rk4.c` can be stored on top of its input; and (3) `rk4.c` does *not* increment `t`, so an explicit incrementation will be necessary before a subsequent call to `rk4.c` in a loop.

A possible driving program to use subroutine `der` (which *we* have written) and subroutine `rk4.c` (supplied in the *Numerical Recipes* library) would have three major sections. First we dimension variables, assign values to the parameters, and initialize variables with the statements\footnote{Since indices in C start at 0 and we wish to have, for example, `n[1]`, `n[2]`, and `n[3]` available, we must dimension `n` at 4 and then ignore the zeroth element `n[0]`.}

```c
34
```
float n[3+1], dndt[3+1]; /* For variables and derivatives */
float t, dt; /* For time, time step */
kA = 0.1; kB = 0.1; /* Set parameters */
dt = 0.25; /* Set time step */
n[1] = 1000.0; n[2] = 0.0; /* Initialize dependent variables */
n[3] = 0.0;
t = 0.0; /* Set initial time */

Then, we open a file to receive the output and write both a labeling first line and the initial values to that file with the statements

fptr = fopen( "decayrk4_c.dat", "w" );
fprintf( fptr, " t A B C
" );
fprintf( fptr, "%6.2f %10.3f %10.3f %10.3f
", t, n[1], n[2], n[3] );

Finally, we execute a loop in which, for each pass, the derivatives at the start of the current interval are evaluated, rk4.c is called to advance the solution (placing its output on top of its input to facilitate the next call), the time is incremented, and the new values are added to the growing file. The statements completing the calculation, including closing the file at the end, are

int i; /* For loop index */
for( i=1; i<=200; i++ )
{
    der( t, n, dndt );
    rk4( n, dndt, 3, t, dt, n, der);
    t = t + dt;
    fprintf( fptr, "%6.2f %10.3f %10.3f %10.3f
", t, n[1], n[2], n[3] );
}
fclose( fptr );

A full listing of this program, which is named decayrk4.c and includes inclusion of necessary header files (both standard C and those specific to Numerical Recipes) and variable declarations, will be found in Section 11.E.

For the sake of an explicit example, we have chosen above to address three-species radioactive decay. Starting from scratch with the intention to use rk4.c, we obtain the actual solution and store it in the file decayrk4_c.dat by

1. Creating the driving program decayrk4.c and the subroutine der using an available text editor (or copying the files from the directory $HEAD/cc).
2. Copying the recipe rk4.c from the directory $NRHEAD/recipes_c-ansi/recipes to the default directory.
3. Copying the needed Numerical Recipes header files nr.h and nrutil.h from the directory $NRHEAD/recipes_c-ansi/include and the needed Numerical Recipes utility subroutine nrutil.c from the directory $NRHEAD/recipes_c-ansi/recipes to the default directory.
4. Compiling and linking the executable module with the statement

   cc -o decayrk4.xc decayrk4.c rk4.c nrutil.c

5. Running the program with the statement ./decayrk4.xc.
Table 11.8: The first and last few lines in the file decayrk4_c.dat.

<table>
<thead>
<tr>
<th>t</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1000.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>0.25</td>
<td>975.310</td>
<td>24.383</td>
<td>0.307</td>
</tr>
<tr>
<td>0.50</td>
<td>951.229</td>
<td>47.561</td>
<td>1.209</td>
</tr>
<tr>
<td>0.75</td>
<td>927.743</td>
<td>69.581</td>
<td>2.676</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>49.50</td>
<td>7.083</td>
<td>35.063</td>
<td>957.854</td>
</tr>
<tr>
<td>49.75</td>
<td>6.909</td>
<td>34.370</td>
<td>958.722</td>
</tr>
<tr>
<td>50.00</td>
<td>6.738</td>
<td>33.690</td>
<td>959.572</td>
</tr>
</tbody>
</table>

The first and last few lines in the file decayrk4_c.dat are shown in Table 11.8.

We conclude this section by illustrating how to construct a driving program to use the stepper routine rkqs.c. The derivatives will, of course, be supplied by the same subroutine that we used with rk4.c. Invocation of rkqs.c, however, requires a bit more preparation. To call rkqs, we use a statement of the form\(^{37}\)

\[
\text{rkqs( } \text{y, dydt, neqs, \&t, dttry, rtol, ys, \&dtact, \&dtnext, drvs )}
\]

where \(y\) and \(dydt\) are vectors containing the values of the dependent variables and their derivatives at time \(t\), \(\text{neqs}\) is the number of equations in the system, \(\text{dttry}\) is the amount by which the independent variable is to be advanced, \(\text{rtol}\) is the desired relative tolerance, \(\text{ys}\) is a vector containing the values relative to which the tolerance is to be assessed, \(\text{dtact}\) is the actual step size on which the adaptive process ultimately settled, \(\text{dtnext}\) is the recommended step size for the next step in the solution, and \(\text{drvs}\) is the name of the subroutine that is to be invoked to calculate the derivatives. Note (1) that rkqs.c requires the current values of the derivatives as input, so an explicit call to the subroutine returning those derivatives will have to precede the call to rkqs.c; (2) in contrast to rk4.c, rkqs.c returns the new values in the variables \(y\) and \(t\), which were used to supply the original values as input.

A driving program to exploit rkqs is similar to—but not identical to—decayrk4.c. The variables storing the parameters must still be declared as global variables by proper positioning of the statement declaring them. Dimensioning variables, setting parameters, and initializing variables is accomplished with the statements listed in Table 11.9. Then, we open a file to receive the output and write both a labeling first line and the initial values to that file with the coding

\[
fptr = fopen( "decayrkqs_c.dat", "w" );
\]

\[
\text{fprintf( fptr, " t A B C\n" );}
\]

\[
\text{fprintf( fptr, "%6.2f %10.3f %10.3f %10.3f\n", t, n[1], n[2], n[3] );}
\]

Finally, we execute a loop in which, for each pass, the derivatives at the start of the current interval are evaluated, rkqs.c is called to advance the solution (placing its output on top of its input to facilitate the next call), the time is incremented, and the new values are added to the growing file. The statements completing the calculation, including a closing of the file at the end, are

\(^{37}\)As illustrated, three of the arguments to rkqs.c must be flagged with a preceding ampersand \& to be properly recognized by the routine.
11.15. SOLVING ODES NUMERICALLY WITH C

Table 11.9: First statements for C Program decayrkqs.c.

```c
float n[3+1], dndt[3+1]; /* For variables and derivatives */
float rel; /* For error control */
float t, dttry; /* For time, time steps */
kA = 0.1; kB = 0.1; /* Set parameters */
dttry = 0.25; /* Set time step */
n[1] = 1000.0; n[2] = 0.0; /* Initialize dependent variables */
n[3] = 0.0;
t = 0.0; /* Set initial time */
rel = 1.0e-6; /* Set desired relative error */
ans[1] = 1000.0; /* Set scaling reference */
ans[2] = 1000.0;
ans[3] = 1000.0;
```

```c
float dtnext, dtact; /* For time steps */
float ans[3+1]; /* For error control */
while ( t <= 50.0 )
{
    der( t, n, dndt );
    rkqs(n, dndt, 3, &t, dttry, rel, ans, &dtact, &dtnext, der );
    fprintf( fptr, "%6.2f %10.3f %10.3f %10.3f\n", t, n[1], n[2], n[3] );
    dttry = dtnext;
}
fclose( fptr );
```

A full listing of this program, which is named decayrkqs.c and includes inclusion of necessary header files (both standard C and those specific to Numerical Recipes) and variable declarations, will be found in Section 11.F.

Starting from scratch with the intention to use rkqs, we obtain the actual solution and store it in the file decayrkqs_c.dat by

1. Creating the driving program decayrkqs.c and the subroutine der using an available text editor (or copying the files from the directory $HEAD/cc).

2. Copying any necessary recipes to the default directory. In the present case, the header files nr.h and nrutil.h, and the utility subroutine nrutil.c presumably remain from the first example. The recipes rkqs.c and rkck.c (whose importance will be proclaimed if the next step is attempted without it) must also be copied from the directory $NRHEAD/recipes_c-ansi/recipes.

3. Compiling and linking the executable module with the statement

   ```bash
   cc -o decayrkqs.xc decayrkqs.c rkqs.c rkck.c nrutil.c -lm
   ```

4. Running the program with the statement ./decayrkqs.xc.

The first and last few lines in the file decayrkqs_c.dat are shown in Table 11.10. Note, in particular, that—because the underlying algorithm is adaptive—this approach has generated solutions at an irregularly spaced sequence of times.
Table 11.10: The first and last few lines in the file decayrkqs_c.dat.

<table>
<thead>
<tr>
<th>t</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1000.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>0.25</td>
<td>975.310</td>
<td>24.383</td>
<td>0.307</td>
</tr>
<tr>
<td>1.35</td>
<td>873.693</td>
<td>117.971</td>
<td>8.336</td>
</tr>
<tr>
<td>3.49</td>
<td>705.048</td>
<td>246.407</td>
<td>48.545</td>
</tr>
<tr>
<td>41.79</td>
<td>15.313</td>
<td>63.993</td>
<td>920.694</td>
</tr>
<tr>
<td>47.27</td>
<td>8.851</td>
<td>41.841</td>
<td>949.307</td>
</tr>
<tr>
<td>53.69</td>
<td>4.658</td>
<td>25.009</td>
<td>970.333</td>
</tr>
</tbody>
</table>

Table 11.11: The first and last few lines in the file decayrkqs_c.dat when DTTRY is adjusted.

<table>
<thead>
<tr>
<th>t</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1000.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>0.25</td>
<td>975.310</td>
<td>24.383</td>
<td>0.307</td>
</tr>
<tr>
<td>0.50</td>
<td>951.229</td>
<td>47.561</td>
<td>1.209</td>
</tr>
<tr>
<td>0.75</td>
<td>927.743</td>
<td>69.581</td>
<td>2.676</td>
</tr>
<tr>
<td>49.75</td>
<td>6.909</td>
<td>34.370</td>
<td>958.722</td>
</tr>
<tr>
<td>50.00</td>
<td>6.738</td>
<td>33.690</td>
<td>959.572</td>
</tr>
<tr>
<td>50.25</td>
<td>6.572</td>
<td>33.022</td>
<td>960.406</td>
</tr>
</tbody>
</table>

If we deleted the statement `dttry = dtnext;` from the loop in decayrkqs.c, the result would be an adaptive incrementation by $\Delta t = 0.25$ with each step and we would have a (longer) table with time values equally spaced. The first and last few lines of that table would be as shown in Table 11.11. These values should be compared with similar tables generated by other methods and presented elsewhere in this chapter. If we believe the program has actually achieved the error tolerance specified with the variable `rel` ($= 10^{-6}$), we should be able to accept these values as accurate to ±0.001.

11.16 Exercises

11.16.1 ... using Symbolic Methods

11.1. Find the motion of a driven, damped oscillator satisfying the differential equation

$$m \frac{d^2 x}{dt^2} + b \frac{dx}{dt} + kx = mf \cos \omega t$$

subject to the general initial conditions

$$x(0) = x_0 ; \quad \frac{dx}{dt}(0) = v_0$$

and then determine the particular initial values that might be imposed so that the transient part of the solution is wiped out from the beginning, i.e., so that the motion is identically the steady-state motion from the moment the oscillator is set into motion. Assume that the oscillator is underdamped. (Note that, so that $m$ will ultimately appear only in conjunction with $b$ and $k$, we have chosen to define $f$ as a force per unit mass rather than simply a force.)
11.2. Take away the walls and the two springs connecting the blocks to the walls in the system of Fig. 11.5, let the masses be different \((m_1 \text{and } m_2, \text{say})\) and denote the constant of the one spring by \(k\). Suppose the blocks are constrained to move along a straight line. Measuring from an arbitrarily selected origin, let the coordinates of the particles be \(x_1\) and \(x_2\), respectively. The equations of motion for this system are

\[
m_1 \frac{d^2x_1}{dt^2} = k(x_2 - x_1) \quad ; \quad m_2 \frac{d^2x_2}{dt^2} = -k(x_2 - x_1)
\]

Let the system be put into motion with arbitrary initial conditions

\[
x_1(0) = x_{10} \quad ; \quad x_2(0) = x_{20} \quad ; \quad \frac{dx_1(0)}{dt} = v_{10} \quad ; \quad \frac{dx_2(0)}{dt} = v_{20}
\]

Solve this initial-value problem for \(x_1(t)\) and \(x_2(t)\) and then examine the behavior of the particular quantities

\[
X(t) = \frac{m_1x_1(t) + m_2x_2(t)}{m_1 + m_2} \quad \text{and} \quad Y(t) = x_2(t) - x_1(t)
\]

which are, respectively, the position of the center of mass of the system and the position of the second block relative to the first block.

11.3. Study the behavior of the system that results when the system shown in Fig. 11.5 is extended to contain three objects coupled in a line. Take the four springs all to have the same spring constant but allow for the possibility that the middle object may have a mass different from that of the two outside objects. Your examination should include at least the following:

- Find both the natural frequencies of the normal modes of oscillation and the initial conditions that will excite the system exclusively in each of the normal modes, determining particularly the frequencies as a function of the ratio of the two masses. To display your results, draw a diagram something like a quantum energy level diagram that shows the way the three natural frequencies vary as the middle mass goes from being rather smaller than the two outer masses to being rather larger than the outer two masses. As initial conditions, it will be sufficient to release the objects from rest at individually arbitrary displacements. You are, however, quite likely to have to help with the inverse Laplace transforms, since your symbolic manipulator may be unable to deal with the inverse Laplace transform of an expression like

\[
\frac{As + Bs^3}{\alpha s^4 + \beta s^2 + \gamma} = s \left( \frac{A + Bs^2}{\alpha s^4 + \beta s^2 + \gamma} \right)
\]

The denominator looks quartic in \(s\) but is better seen as quadratic in \(s^2\). If you introduce \(z = s^2\) temporarily, factor the resulting denominator, and then do a partial fraction expansion, you can end up with the alternative form

\[
s \left( \frac{A + Bs^2}{\alpha s^4 + \beta s^2 + \gamma} \right) = s \left( \frac{R}{s^2 + R^2} + \frac{Q}{s^2 + Q^2} \right) = \frac{Rs}{s^2 + R^2} + \frac{Qs}{s^2 + Q^2}
\]

whose inverse Laplace transform is easier to find. (Indeed, even Table 1.2 is adequate to the task.) Finding the constants \(R, A, R'\) and \(Q'\) is part of the exercise; the expressions are not pretty. The lesson: Symbolic manipulators frequently need help from someone who really knows where the calculation is going.

- With the three masses equal, find the solution for \(x_1(t)\), \(x_2(t)\), and \(x_3(t)\) when one of the outer masses is initially displaced, the other two are not, and all three are released from rest. Plot graphs of all three positions as functions of time over a long enough time interval to reveal the features of the motion.

- With the three masses equal, find the solution for \(x_1(t)\), \(x_2(t)\), and \(x_3(t)\) when the middle mass is initially displaced, the other two are not, and all three are released from rest. Plot graphs of all three positions as functions of time over a long enough time interval to reveal the features of the motion.
To help you get started and to facilitate focusing on the solution of the ODEs rather than on deriving them, note that, for three masses, the equations of motion will be

\[ m \frac{d^2 x_1}{dt^2} = -kx_1 + k(x_2 - x_1) \]
\[ m \frac{d^2 x_2}{dt^2} = -k(x_2 - x_1) + k(x_3 - x_2) \]
\[ m \frac{d^2 x_3}{dt^2} = -k(x_3 - x_2) - kx_3 \]

11.4. The system called the double pendulum shown in Fig. 11.7 consists of a ball of mass \( m_1 \) hanging from a rigid and massless rod of length \( l_1 \) attached to the ceiling and a second ball of mass \( m_2 \) hanging from a rigid and massless rod of length \( l_2 \) attached to the first ball. The balls swing in a plane, and the configuration of the system is specified by giving two angles, the first of which, \( \theta \), gives the angle that the upper string makes with the vertical and the second of which, \( \phi \), gives the angle that the lower string makes with the vertical. The motion can be very complicated and at times will be chaotic. For small amplitudes, however, things are much more sedate. When the amplitudes of the motion of both balls are small and—to simplify a little bit—when the strings are both the same length (\( l_1 = l_2 \), which we will symbolize with the letter \( l \)), the equations of motion turn out to be

\[ \frac{d^2 \theta}{dt^2} + \frac{m_2}{m_1 + m_2} \frac{d^2 \phi}{dt^2} + \frac{g}{l} \theta = 0 \]
\[ \frac{d^2 \phi}{dt^2} + \frac{d^2 \theta}{dt^2} + \frac{g}{l} \phi = 0 \]

Find the normal modes of oscillation of this system and determine the initial conditions that will cause the system to oscillate exclusively in one or the other of these modes.

11.5. Solve the differential equation

\[ m \frac{d^2 x}{dt^2} + b \frac{dx}{dt} + kx = mf \cos \omega t \]

for the driven, damped harmonic oscillator subject to the specific initial conditions

\[ x(0) = x_0 ; \quad \frac{dx}{dt}(0) = 0 \]

by explicitly taking the Laplace transform of the equation, solving that result for the Laplace transform of the solution, and then inverting that transform to obtain the solution itself.

11.6. Among the simplest of differential equations is the equation

\[ \frac{d^2 x}{dt^2} + \omega^2 x = 0 \]

that describes a simple harmonic oscillator. Generate a series solution to this equation and then verify that the solution thus generated agrees with the known solution

\[ x(t) = A \cos \omega t + B \sin \omega t \]
11.7. Find a symbolic solution for all three components for the motion of a projectile in a linear, viscous medium, when the initial conditions are general, i.e., solve Eq. (11.4) subject to the initial conditions in Eq. (11.2). Since the equations are uncoupled, you can solve each individually. Alternatively, you can solve the three equations simultaneously as a system. Solve them both ways. Once you have the solutions in hand, verify that they satisfy the original equations and initial conditions. Finally, explore their limits for small \( b \).

11.8. Recast Eq. (11.34) for the LRC circuit in dimensionless form, measuring charge in units of \( q_0 \) and finding a suitable unit in terms of which to measure time.

11.9. Reflecting Coulomb’s law, the equation of motion for a particle carrying charge \( q \) moving in the (fixed) electrostatic field of a charge \( Q \) is

\[
m \frac{d^2 \mathbf{r}}{dt^2} = \frac{qQ}{4\pi\epsilon_0 r^3}
\]

Here, the force center is assumed to be at the origin, \( \mathbf{r} \) is the position vector of the particle, and the equation is written in mksa units. Write this equation of motion out in Cartesian coordinates, cast it in dimensionless form, and determine the correspondences that must be adopted to convert the equations for an orbit in an electrostatic field into those for the orbit in a gravitational field, i.e., into Eq. (11.46).

11.10. Recast the system of six first-order equations in the last paragraph of Section 11.1.1 in dimensionless form, both when the viscous term is linear in the velocity and when it is quadratic in the velocity. Appropriate units for this recasting can be inferred from the discussion earlier in that section.

11.11. Translate the entire discussion—both dimensional and dimensionless—of the planetary problem in Section 11.1.7 from Cartesian coordinates \((x, y)\) to polar coordinates \((r, \phi)\), where \( x = r \cos \phi \) and \( y = r \sin \phi \). That is

(a) Show that the polar components of the equations of motion in the first instance are

\[
m \left[ \frac{d^2 r}{dt^2} - r \left( \frac{d\phi}{dt} \right)^2 \right] = f(r) ; \quad m \left( \frac{d^2 \phi}{dt^2} + \frac{d\phi}{dt} \frac{dr}{dt} \right) = 0
\]

(b) Recognizing that

\[
\frac{d}{dt} \left( mr^2 \frac{d\phi}{dt} \right) = m \left( r^2 \frac{d^2 \phi}{dt^2} + 2r \frac{dr}{dt} \frac{d\phi}{dt} \right)
\]

show that

\[
m r^2 \frac{d\phi}{dt} = \text{constant} = L \quad \Rightarrow \quad \frac{d\phi}{dt} = \frac{L}{mr^2}
\]

and then that

\[
m \frac{d^2 r}{dt^2} = f(r) + \frac{L^2}{mr^3}
\]

Here, \( L = m(x_0 v_{y0} - y_0 v_{x0}) \) is the angular momentum of the object; \( L \) is constant throughout the motion. (In polar coordinates, we would first solve this single, second-order, non-linear, inhomogeneous equation for \( r(t) \). Then, with \( r(t) \) in hand, we integrate the equation \( d\phi/dt = L/mr^2 \)—see Chapter 13—to find \( \phi(t) \).

(c) Translate the initial conditions to polar coordinates, finding that

\[
r(0) = \sqrt{x_0^2 + y_0^2} \quad \frac{dr}{dt}(0) = \frac{x_0 v_{x0} + y_0 v_{y0}}{\sqrt{x_0^2 + y_0^2}}
\]

\[
\phi(0) = \arctan \frac{y_0}{x_0} \quad \frac{d\phi}{dt}(0) = \frac{x_0 v_{y0} - y_0 v_{x0}}{x_0^2 + y_0^2}
\]
(d) Restrict the force to the inverse square gravitational force, finding that
\[ \frac{d^2 r}{dt^2} = -G \frac{M}{r^2} + \frac{L^2}{m^2 r^3} \]

(e) Recast the differential equations and initial conditions in dimensionless form, finding that
\[ \frac{d^2 r}{dt^2} = -\frac{1}{\overline{r}^2} + \frac{\beta}{\overline{r}^3} ; \quad \frac{d\phi}{dt} = \pm \sqrt{\beta} \overline{r} \]
\( r(0) = \frac{r_0}{\ell} = \sqrt{x_0^2 + y_0^2} / \ell ; \quad \phi(0) = \arctan \frac{y_0}{x_0} \)
\[ \frac{dr}{dt}(0) = x_0 v_{x0} + y_0 v_{y0} \sqrt{x_0^2 + y_0^2} / \sqrt{GM/\ell} ; \quad \frac{d\phi}{dt} = \frac{x_0 v_{y0} - y_0 v_{x0}}{(x_0^2 + y_0^2) \sqrt{GM/\ell}^3} \]
where \( \overline{\beta} = \frac{L^2}{(GMm^2\ell)} \) is a constant, and—in the second equation—the upper sign applies when \( L > 0 \) and the lower sign when \( L < 0 \). While all parameters disappeared from the equations in Cartesian coordinates, the parameter \( \beta \) remains in the equations in polar coordinates.

(f) Recast the Cartesian expressions in the last paragraph of Section 11.1.7 for both the dimensional and the dimensionless statements of conservation of energy and angular momentum in the planetary problem into polar coordinates.

Hint: Remember (or take as given if you haven’t met them yet) that the radial and azimuthal components of the acceleration of a particle in polar coordinates are given by \( a_r = \ddot{r} - r \dot{\phi}^2 \) and \( a_\phi = r \ddot{\phi} + 2 \dot{r} \dot{\phi} \).

11.16.2 . . . using Numerical Methods

11.12. In an appropriate dimensionless presentation, standing waves in a string must satisfy the boundary value problem
\[ \frac{d^2 y}{dx^2} + k^2 y = 0 ; \quad y(0) = y(1) = 0 \]
Suppose that the interval \( 0 \leq x \leq 1 \) is divided into \( n \) equal segments of length \( \Delta x = 1/n \), let \( x_i = i \Delta x \) (with \( i = 0, 1, 2, \ldots, n \)), and let \( y_i = y(x_i) \). Evaluate the ODE at \( x = x_i \), approximate the second derivative with the difference formula
\[ \frac{d^2 y}{dx^2} \bigg|_{x=x_i} \approx \frac{y_{i+1} - 2y_i + y_{i-1}}{\Delta x^2} ; \quad i = 1, 2, 3, \ldots n - 1 \]
and note that \( y_0 = y_n = 0 \). Show that the values \( y_i \) for \( i = 0, 1, 2, \ldots, n \) satisfy a system of \( n + 1 \) linear algebraic equations of the form \( M Y = \alpha Y \), where \( Y \) is an \( (n+1) \)-component vector whose elements are the values of \( y_i \) and \( \alpha \) is determined from \( k^2 \) and \( \Delta x \). Then argue that the allowed values of \( k^2 \) can be determined from the eigenvalues of the matrix \( M \). That is, show that this transformation turns a boundary value problem involving a differential equation into an approximately equivalent matrix eigenvalue problem.

11.14. Some radioisotopes exhibit a branched decay sequence such as

A \[ \begin{array}{c} \uparrow \end{array} \]
B \[ \begin{array}{c} \uparrow \end{array} \]
C \[ \begin{array}{c} \uparrow \end{array} \]
D

\[ k_1 \]
\[ k_2 \]
\[ k_3 \]
\[ k_4 \]

Deduce appropriate differential equations, create an appropriate file to define the equations, and thoroughly explore the behavior of this system. Assume that \( D \) is stable and that, initially, only \( A \) is present.
11.15. Taking the initial conditions for the unforced, damped harmonic oscillator of Eq. (11.27) with \( \mathcal{F} = 0 \) to be \( \mathcal{F}(0) = 1 \) and \( \mathcal{F}(0) = 0 \), produce graphs of \( \mathcal{F} \) versus \( t \) for several values of \( \beta \) on the interval \( 0.0 < \beta < 2.0 \). Then using the knowledge that \( \beta = \frac{b}{m\omega} = \frac{b}{\sqrt{mk}} \), and that \( t = \omega t \), infer from your graphs the effect of changing \( m \) or \( k \) on the physical motion.

11.16. Explore the unforced, damped harmonic oscillator of Eq. (11.27) with \( \mathcal{F} = 0 \) for the cases of critical damping (\( \beta = 2.0 \)) and overdamping (\( \beta > 2.0 \)).

11.17. Explore the behavior of the Van der Pol oscillator described in dimensionless form by the equation
\[
\frac{d^2\theta}{dt^2} + \frac{\omega}{l} \sin \theta = 0
\]
where \( \theta \) is measured in radians from the lowest point of the pendulum’s motion. Use numerical methods to study the motion of this pendulum when it is released from rest at each of several initial displacements, say 20°, 45°, 90°, 120°, 150°, 165°, and 178°. Look particularly at graphs of \( \theta \) versus \( t \), \( d\theta/dt \) versus \( t \), and \( d^2\theta/dt^2 \) versus \( \theta \) (the phase plot). Obtain also a graph of period versus amplitude (initial displacement). Write several paragraphs describing your set up of the problem and presenting evidence for your discoveries. Optional: Try starting the pendulum at the bottom (0 initial angle) with several different initial angular velocities. How large can the angular velocity be before the pendulum begins to swing over the top? Suggestion: Begin by introducing the dimensionless time \( t = \sqrt{g/l} t \) so that the equation becomes \( d^2\theta/dt^2 + \sin \theta = 0 \).

11.18. Suppose that the “gravitational” force were not inverse square but instead depended on some other (negative) power of the radial coordinate. The dimensionless equations of motion then would be
\[
\frac{d^2\xi}{dt^2} = -\frac{\xi}{(\xi^2 + \eta^2)^b} \quad ; \quad \frac{d^2\eta}{dt^2} = -\frac{\eta}{(\xi^2 + \eta^2)^b}
\]
[Compare Eq. (11.46).] Of course, the equations reduce to those for the inverse square force if we simply set \( b = 3/2 \). For the planetary problem, find conditions that will generate a distinctly elliptical orbit for an attractive inverse square force (\( b = 1.5 \)). Then explore the effect on that orbit of distorting the force by changing the exponent in the denominator of the equations making \( b = 1.45 \), \( b = 1.55 \), or anything else you can think of, and write a paragraph or two describing the nature of the changes in some detail. Make sure your solutions are generated to an adequate accuracy to support your conclusions.

11.19. Explore the scattering orbits that occur when an object moves under the action of a repulsive inverse square force, and compare your results with those for an attractive force.

11.20. Deduce the equations of motion for a space ship of mass \( m \) coasting freely in the \( xy \) plane under the gravitational influence of two suns, each of mass \( M \) and located respectively at \((R,0)\) and \((-R,0)\). Then express the equations in dimensionless form and, creating all necessary files, thoroughly explore the motion of this space ship. In particular, you might search for an orbit that loops like a figure-eight around the two suns and/or you might see if your approach predicts what you would expect intuitively if you start the spaceship from rest at a point on the perpendicular bisector of the line joining the two suns. Make sure your solutions are generated to an adequate accuracy.

11.22. Suppose a particle of charge \( q \) and mass \( m \) is injected into a region of space containing constant, crossed electric and magnetic fields \( \mathbf{E} = E_x \hat{i} \) and \( \mathbf{B} = B_x \hat{k} \). In vector form, the equation of motion for this particle is
\[
m \frac{d^2\mathbf{r}}{dt^2} = q \mathbf{E} + q \frac{d\mathbf{r}}{dt} \times \mathbf{B}
\]
CHAPTER 11. SOLVING ORDINARY DIFFERENTIAL EQUATIONS

Verify the equations of motion

\[ m \frac{d^2x}{dt^2} = qE_x + qB_z \frac{dy}{dt} ; \quad m \frac{d^2y}{dt^2} = -qB_z \frac{dx}{dt} ; \quad m \frac{d^2z}{dt^2} = 0 \]

for the specific fields of this exercise, express them in dimensionless form (note that \( \omega = qB_z/m \) is a frequency and \( E_x/B_z \) is a velocity), and thoroughly explore the behavior of the particle in this situation. Try to understand the motion intuitively. \( \text{Hint:} \) You should find that, in terms of an arbitrarily selected unit of length \( \ell \), the equations involve a single parameter \( \frac{qE_x}{(m\omega^2\ell)} \), which can alternatively be written as \( \frac{(E_x/B_z)}{(\omega \ell)} \) — the ratio of the velocity \( E_x/B_z \) determined by the fields to the characteristic velocity implied by your choice of a length unit and the frequency \( \omega \). Note that this exercise actually has more than one parameter, since the initial components of the velocity—probably expressed in units of \( \omega \ell \)—also influence the solution.

11.23. A particle having mass \( m \) and carrying charge \( q \) moves in the \( xy \)-plane while experiencing an electric field given by \( \mathbf{E}(x,y) = -\alpha y \hat{j} \), where \( \alpha \) is a constant. Assume that \( \alpha \) and \( q \) are both positive. With \( \mathbf{r} = x \hat{i} + y \hat{j} \), the vector equation of motion for this particle then is

\[ m \frac{d^2\mathbf{r}}{dt^2} = -q\alpha y \hat{j} \]

(a) Show that, in component form, the equations of motion for this particle are

\[ \frac{d^2x}{dt^2} = 0 ; \quad \frac{d^2y}{dt^2} = -\frac{q\alpha}{m} y = -by \]

(b) With \( b \) a global parameter, create an appropriate file to define these equations for the ODE-solver in an available numerical/graphical tool. (c) Use that tool to obtain graphs of the trajectories in the \( xy \)-plane of several particles projected from the origin at different angles and with different speeds. (d) Speculate on a use for this field.

11.24. An important system in the early study of chaos is described by the Lorenz equations

\[
\begin{align*}
\frac{dx}{dt} &= a(y - x) \\
\frac{dy}{dt} &= -xz + bx - y \\
\frac{dz}{dt} &= xy - cz
\end{align*}
\]

Create an appropriate file defining these equations and then thoroughly explore the behavior of this system. Graphs of \( y \) versus \( x \), \( z \) versus \( x \), and \( z \) versus \( y \) when \( a = 10.0, b = 28.0 \) and \( c = 8.0/3.0 \) under the initial conditions \( x_0 = 1.0, y_0 = 0.0, \) and \( z_0 = 0.0 \) are particularly interesting. While graphs of \( y \) versus \( x \), \( z \) versus \( x \), and \( z \) versus \( y \) are interesting, the true beauty of the trajectory is best seen using a three-dimensional space curve. Be sure to examine the path from several different vantage points in the space around the path, an objective most easily accomplished if the display of the path allows rotation of the path on the screen.

11.26. Recast the program \texttt{decay.c} so that it invokes (a) the improved Euler method and (b) the second-order Runge-Kutta method rather than Euler’s method to solve the problem of three-species decay. Then, compile and test your programs. Give particular attention to exploring the accuracy of the solution by using several different time steps. Further, compare the accuracy obtained for various time steps with that obtained for the same time steps using Euler’s method.

11.27. The dynamics of the chemical reaction

\[ A + B \rightleftharpoons C + D \]

is governed by the equations

\[
\begin{align*}
\frac{dA}{dt} &= -k_f AB + k_r CD \\
\frac{dB}{dt} &= -k_f AB + k_r CD \\
\frac{dC}{dt} &= k_f AB - k_r CD \\
\frac{dD}{dt} &= k_f AB - k_r CD
\end{align*}
\]
where \( A(t) \), \( B(t) \), \( C(t) \), and \( D(t) \) are the concentrations of each molecule in the reaction vessel, and \( k_f \) and \( k_r \) are the forward and reverse rate constants, respectively. Suppose that the reaction is started with \( A(0) = A_0 \), \( B(0) = B_0 \), and \( C(0) = D(0) = 0 \). Cast the equations in dimensionless form, using \( A_0 \) as the unit of concentration and \( k_f A_0 t \) as the dimensionless time. Then explore the behavior of the system as a function of the initial concentration of \( B \), measured in units of \( A_0 \) and the reverse rate constant, measured in units of \( k_f \). Look particularly at the dependence of the ultimate equilibrium on these parameters. Make sure your results are generated to adequate accuracy.

11.28. In classical ecology, the interaction between a predator and a prey, with populations \( x(t) \) and \( y(t) \), respectively, is modeled with the equations
\[
\frac{dx}{dt} = -k_1 x + k_2 xy ; \quad \frac{dy}{dt} = k_3 y - k_4 xy
\]
where \( k_1 \) and \( k_3 \) are parameters describing the way each population would evolve in the absence of the other and \( k_2 \) and \( k_4 \) are parameters describing strength of the interaction between the two species, which we take to be proportional to the likelihood of an encounter between a member of one species and a member of the other species. Depending on the parameters and the initial populations \( x(0) = x_0 \), \( y(0) = y_0 \), the system may approach a stable equilibrium or, alternatively, one or the other of the populations may become extinct. Explore this system to determine conditions under which each of these circumstances occurs and write a paragraph or two describing your findings. Make sure your results are generated to an accuracy adequate to support your conclusions.

11.29. Examine the undamped, unforced harmonic oscillator carefully, using at least two different methods and several time steps. Monitor the accuracy of your solution by monitoring the constancy of the energy of the oscillator given (in dimensional form) by \( E = \frac{1}{2}mv^2 + \frac{1}{2}kx^2 \).

11.30. In the text, we illustrated the use of one or more numeric processing programs to solve the chain radioactive decay problem of Section 11.1.2 using Euler’s method. Using an available numeric processing program like IDL, MATLAB, OCTAVE, or PYTHON, repeat that development using the improved Euler method and comment on what the resulting graphs reveal about the adequacy of the several time steps used.

11.31. The type of child’s swing shown in Fig. 11.8 is hung with elastic ropes. Suppose the ropes are long enough so that the child/swing can be represented by a point mass at the end of a spring, assume the spring obeys Hooke’s law with constant \( k \) and has an unstretched length \( a \), and let the motion of the mass be confined to a single vertical plane. Show that, in the coordinate system illustrated, the equations of motion are
\[
m \frac{d^2x}{dt^2} = -kx + \frac{kax}{\sqrt{x^2 + y^2}} ; \quad m \frac{d^2y}{dt^2} = mg - k_4y + \frac{kay}{\sqrt{x^2 + y^2}}
\]
Then, introducing \( \omega_0^2 = k/m, \bar{t} = \omega_0 t, \bar{x} = x/a, \bar{y} = y/a \), cast the equations in dimensionless form. After the suggested rescalings, only one parameter—\( g/\omega_0^2 \)—remains [and this parameter is the square of the ratio of the frequency \( \sqrt{g/a} \) of a simple pendulum of length \( a \)—call it the ”swing” frequency—to the frequency \( \sqrt{k/m} \) of an object of a mass \( m \) bobbing on a spring of stiffness \( k \)—call it the ”bounce” frequency. Using an available numeric processing program like IDL, MATLAB, OCTAVE, or PYTHON, explore the motions for several values of this one parameter, including values larger than, equal to, and smaller than 1. Write several paragraphs describing and presenting evidence for your discoveries.

11.16.3 . . . using Numerical Recipes

Note: Numerical recipes can, of course, also be used for any of the exercises in Section 11.16.2.

11.32. Read the book Numerical Recipes to discover how to use the routine odeint and then use it to study three-species radioactive decay.
11.33. Read the book *Numerical Recipes* to discover how to use the routine *mmid* and then use it to study three-species radioactive decay.

11.34. Use two different numerical recipes of your choice to study the logistic growth of a population.

11.35. Study the damped, unforced harmonic oscillator using (a) *rk4* and (b) *rkqs*. Consider under-damped, critical, and overdamped cases.

11.36. Use (a) *rk4* and (b) *rkqs* to study both bound and unbound orbits in the inverse square gravitational force.

11.37. Choose one of the exercises from Section 11.16.2 and address it using at least one numerical recipe.
11.D Listing of decay.c

/* PROGRAM decay.c */

#include <stdio.h>
#include <math.h>

main()
{

/***** Declare variables *****/

FILE *fptr;                /* For file pointer */
float kA, kB;               /* For parameters */
float t, A, B, C;          /* For independent, dependent variables */
float dAdt, dBdt, dCdt;    /* For derivatives */
float dt;                   /* For time increment */
int i, nsteps;             /* Loop index, limit */

/***** Get parameters, time step, initial values *****/

printf( "Decay constant for A: " ); scanf( "%f", &kA );
printf( "Decay constant for B: " ); scanf( "%f", &kB );
printf( "Number of steps : " ); scanf( "%d", &nsteps );
printf( "Time step : " ); scanf( "%f", &dt );
printf( "Initial A : " ); scanf( "%f", &A );
printf( "Initial B : " ); scanf( "%f", &B );
printf( "Initial C : " ); scanf( "%f", &C );
t = 0.0;

/***** Open file, write label and initial values *****/

fptr = fopen( "decay_c.dat", "w" );
fprintf( fptr, " t A B C
" );
fprintf( fptr, "%6.2f %10.3f %10.3f %10.3f
", t, A, B, C );

/***** Solve equations, writing results to file *****/

for( i=1; i<=nsteps; i++ )
{
    dAdt = -kA*A;             /* Calculate current derivatives */
    dBdt = kA*A - kB*B;
    dCdt = kB*B;
    A = A + dAdt*dt;          /* Calculate new values */
    B = B + dBdt*dt;
    C = C + dCdt*dt;
    t = t + dt;
    fprintf( fptr, "%6.2f %10.3f %10.3f %10.3f
", t, A, B, C );
}

/***** Close file *****/

fclose( fptr );
}
CHAPTER 11. SOLVING ORDINARY DIFFERENTIAL EQUATIONS

11.E Listing of decayrk4.c

/* PROGRAM decayrk4.c */

#include <stdio.h>
#include <math.h>
#include "nr.h"
#include "nrutil.h"

float kA, kB; /* For parameters */

void der( float t, float n[], float dndt[] )
{
    dndt[1] = -kA*n[1]; /* Calculate dA/dt */
    dndt[3] = kB*n[2]; /* Calculate dC/dt */
}

main()
{
    FILE *fptr; /* For file pointer */
    float n[3+1], dndt[3+1]; /* For variables and derivatives */
    float t, dt; /* For time, time step */
    int i; /* For loop index */

    kA = 0.1; kB = 0.1; /* Set parameters */
    dt = 0.25; /* Set time step */
    n[1] = 1000.0; n[2] = 0.0; /* Initialize dependent variables */
    n[3] = 0.0; /* Set initial time */

    fptr = fopen( "decayrk4_c.dat", "w" );
    fprintf( fptr, " t A B C\n" );
    fprintf( fptr, "%6.2f %10.3f %10.3f %10.3f\n", t, n[1], n[2], n[3] );

    for( i=1; i<=200; i++ )
    {
        der( t, n, dndt );
        rk4( n, dndt, 3, t, dt, n, der);
        t = t + dt;
        fprintf( fptr, "%6.2f %10.3f %10.3f %10.3f\n", t, n[1], n[2], n[3] );
    }
    fclose( fptr );
}
11.F Listing of decayrkqs.c

/* PROGRAM decayrkqs.c */

#include <stdio.h>
#include <math.h>
#include "nr.h"
#include "nrutil.h"

float kA, kB; /* For parameters */

void der( float t, float n[], float dndt[] )
{
    dndt[1] = -kA*n[1]; /* Calculate dA/dt */
    dndt[3] = kB*n[2]; /* Calculate dC/dt */
}

main()
{
    FILE *fptr; /* For file pointer */
    float n[3+1], dndt[3+1]; /* For variables and derivatives */
    float rel, ans[3+1]; /* For error control */
    float t, dttry, dtnext, dtact; /* For time, time steps */

    kA = 0.1; kB = 0.1; /* Set parameters */
    dttry = 0.25; /* Set time step */
    n[1] = 1000.0; n[2] = 0.0; /* Initialize dependent variables */
    n[3] = 0.0;
    t = 0.0; /* Set initial time */
    rel = 1.0e-6; /* Set desired relative error */
    ans[1] = 1000.0; /* Set scaling reference */
    ans[2] = 1000.0;
    ans[3] = 1000.0;

    fptr = fopen( "decayrkqs_c.dat", "w" );
    fprintf( fptr, "t  A  B  C\n" );
    fprintf( fptr, "%6.2f %10.3f %10.3f %10.3f\n", t, n[1], n[2], n[3] );

    while ( t <= 50.0 )
    {
        der( t, n, dndt );
        rkqs(n, dndt, 3, &t, dttry, rel, ans, &dtact, &dtnext, der );
        fprintf( fptr, "%6.2f %10.3f %10.3f %10.3f\n", t, n[1], n[2], n[3] );
        dttry = dtnext;
    }
    fclose( fptr );
}
Chapter 13

Evaluating Integrals

Frequently the answer to an interesting question in physics is given by—or can be cast in the form of—an integral, often as a function of the upper limit or as a parameter in the integrand. Sometimes, that integral can be evaluated in closed form. More often, however, the integral is analytically intractable and must be approached numerically. We begin this chapter by identifying several physical situations, the full addressing of which requires evaluating an integral, perhaps as a function of one or more parameters. Then we illustrate how to use symbolic algebra systems to approach those that can be evaluated analytically, describe a few of many available numerical algorithms (with attention to their accuracy), and—finally—describe ways to evaluate representative integrals using a variety of numerical approaches and computational tools. When parameters are involved, we also illustrate how to plot graphs of the integrals as functions of those parameters.

We shall classify each integral in one of three categories, since the approach to its numerical evaluation will depend on this classification. Integrals in the first category will simply be a number; their numerical evaluation involves a single invocation of one or another basic algorithm. In more complicated—and more interesting—cases, the integral will be a function of a parameter, which may appear in the limits of integration (second category) or embedded in the integrand (third category); the numerical evaluation of these integrals as a function of the parameter will involve repeated invocation of one or another basic algorithm within a loop.

13.1 Sample Problems

In this section, we identify several physical contexts in which integrals arise, and we determine a representative integral for each case.

13.1.1 One-Dimensional Trajectories

The motion of a particle of mass $m$ in one dimension under the action of a force $f$ satisfies Newton’s second law, equivalent to the two equations

$$
\frac{dp}{dt} = f \quad \text{and} \quad \frac{dx}{dt} = v 
$$

which are to be solved subject to the prescribed initial conditions $x(0) = x_0$, $v(0) = v_0$, and $p(0) = p_0$. Here, $x$, $v$, and $p$, are the position, velocity, and momentum of the particle, respectively, and, with $c$ standing for the speed of light, the relationship between $p$ and $v$ assumes one of the forms

$$
p = \frac{mv}{\sqrt{1 - v^2/c^2}} \quad \text{or} \quad p = mv
$$

(13.2)
depending on whether the motion is relativistic or nonrelativistic. If the force happens to depend only on \( t \), the solution to these two differential equations can be expressed as the explicit integrals

\[
p(t) = p_0 + \int_0^t f(t') \, dt' \quad \text{and} \quad x(t) = x_0 + \int_0^t v(t') \, dt'
\]

(13.3)

and the physical problem of predicting the trajectory reduces to the mathematical problem of evaluating two integrals, finding the momentum from the first integral in Eq. (13.3), then solving for the velocity \( v(t) \) using the appropriate member of Eq. (13.2), and finally finding \( x(t) \) from the second integral in Eq. (13.3). Each integral is a function of at least one parameter—the upper limit \( t \)—and will also depend on additional quantities (e.g., \( m, c, \ldots \)) in the integrand (unless a dimensionless casting happens to suppress them). These integrals fall into either the second or the third of our three categories.

If, on the other hand, the force happens to depend only on \( x \), we can recast the computational task by noting first that

\[
d\frac{d^2x}{dt^2} = \frac{dv}{dt} = \frac{dv}{dx} \frac{dx}{dt} = v \frac{dv}{dx}
\]

(13.4)

Then, Newton’s second law becomes

\[
m \frac{d^2x}{dt^2} = f(x) \quad \Rightarrow \quad m \frac{dv}{dx} = f(x) \quad \Rightarrow \quad m v \frac{dv}{dx} = f(x) \, dx
\]

(13.5)

Finally, by integrating this last expression from initial values to general values at some other time, we find that

\[
m \int_{x_0}^x v' \, dv' = \frac{1}{2} m v^2 - \frac{1}{2} m v_0^2 = \int_{x_0}^x f(x') \, dx'
\]

(13.6)

and the task of finding the velocity (as a function of \( x \)) is reduced to the straight-forward evaluation of an integral. Once that integral has been evaluated, we can then predict the position by exploiting the relationship

\[
\frac{dx}{dt} = v(x) \quad \Rightarrow \quad dt = \frac{dx}{v(x)} \quad \Rightarrow \quad \int_{t_0}^t dt' = t - t_0 = \int_{x_0}^x \frac{dx'}{v(x')}
\]

(13.7)

and we have reduced the second step in the solution to the evaluation of an integral as well. Both integrals deduced in this paragraph may have internal parameters, so they might fall into either the second or the third of our three categories.

13.1.2 Center of Mass

The location \( \mathbf{r}_{\text{cm}} \) of the center of mass of an object is given by

\[
\mathbf{r}_{\text{cm}} = \frac{1}{M} \int \mathbf{r} \, dm
\]

(13.8)

where \( \mathbf{r} \) locates a representative element of the object, \( dm \) is the mass of that element, \( M = \int dm \) is the total mass of the object, and the integral extends over the region of space (volume, surface, or line) occupied by the object. More specifically, if the object lies in a plane and polar coordinates \((r, \phi)\) are appropriate, we would write this integral more explicitly in the form

\[
\mathbf{r}_{\text{cm}} = \frac{1}{M} \int \sigma(r, \phi) \, r \, dr \, d\phi = \frac{1}{M} \int \sigma(r, \phi) \left( r \cos \phi \, \mathbf{i} + r \sin \phi \, \mathbf{j} \right) \, r \, dr \, d\phi
\]

(13.9)

where \( \sigma(r, \phi) \) is the mass per unit area of the object and \( r \, dr \, d\phi \) is the area of the chosen element.
13.1. SAMPLE PROBLEMS

Even more specifically, if we seek the center of mass of the uniform semicircular plate of total mass $M$, radius $a$, and mass per unit area $\sigma = M/(1/2 \pi a^2)$ shown in Fig. 13.1, we would conclude that

$$r_{cm} = \frac{1}{M} \int_0^a \int_0^{\pi} \frac{M}{2 \pi a^2} \left( r \cos \phi \; \hat{i} + r \sin \phi \; \hat{j} \right) r \, dr \, d\phi = \frac{2}{\pi a^2} \int_0^a \int_0^{\pi} r^2 \left( \cos \phi \; \hat{i} + \sin \phi \; \hat{j} \right) dr \, d\phi \quad (13.10)$$

Recognizing as always the wisdom of casting problems to be addressed with a computer in dimensionless form, we finally introduce the dimensionless length $\lambda = r/a$, in terms of which we then find that

$$r_{cm} a = \frac{2}{\pi} \int_0^1 \int_0^{\pi} \lambda^2 \left( \cos \phi \; \hat{i} + \sin \phi \; \hat{j} \right) d\phi \, d\lambda \quad (13.11)$$

Although this integral is two-dimensional, it clearly falls into the first of our three categories—an integral whose value is simply a number (or, in this case, a constant vector).

13.1.3 Moment of Inertia; Radius of Gyration

The moment of inertia $I$ of an object of mass $M$ about a chosen axis is given by

$$I = \int r^2 \, dm \quad (13.12)$$

where $r$ is the distance of an element of the object from the chosen axis, $dm$ is the mass of that element, and the integral extends over the region of space (volume, surface, or line) occupied by the object. Further, the radius of gyration $k$ of this object with respect to the same axis is defined so that a point object of mass $M = \int dm$ located at the distance $k$ from the axis has the same moment of inertia as the object itself, i.e., $k$ is defined so that

$$I = Mk^2 \quad \Rightarrow \quad k = \sqrt{\frac{I}{M}} \quad (13.13)$$

Suppose, for example, we seek the moment of inertia of the uniform semicircular plate of mass $M$ and radius $a$ shown in Fig. 13.1 about the $x$ axis. As in the previous example, the mass per unit area $\sigma$ is given by $\sigma = M/(1/2 \pi a^2)$. We choose a horizontal strip, all elements of which are the same distance $y$ from the $x$ axis. If this strip has mass $dm$, its contribution to the moment of inertia about that axis is $y^2 \, dm$, and the moment of inertia of the plate about that axis is given by

$$I = \int_{y=0}^{y=a} y^2 \, dm \quad (13.14)$$
Remembering that the equation of a circle of radius $a$ is $x^2 + y^2 = a^2$, we note next that the length of the illustrated strip at height $y$ is $2\sqrt{a^2 - y^2}$. If we take the width of that strip to be $dy$, then its area is given by $dA = 2\sqrt{a^2 - y^2} \, dy$, and its mass is given by

$$dm = \sigma \, dA = \frac{4M}{\pi a^2} \sqrt{a^2 - y^2} \, dy$$

(13.15)

Finally, the moment of inertia of the entire object is given by the integral

$$I = \frac{4M}{\pi a^2} \int_0^a y^2 \sqrt{a^2 - y^2} \, dy = \frac{4}{\pi} Ma^2 \int_0^1 \lambda^2 \sqrt{1 - \lambda^2} \, d\lambda$$

$$\Rightarrow \quad \frac{I}{Ma^2} = \frac{4}{\pi} \int_0^1 \lambda^2 \sqrt{1 - \lambda^2} \, d\lambda$$

(13.16)

where we have introduced the dimensionless variable $\lambda = y/a$ and expressed the moment of inertia in units of $Ma^2$, which is the moment of inertia of a point mass $M$ a distance $a$ from the axis. This integral is simply a number, and it therefore falls into the first of our three categories.

### 13.1.4 The Large Amplitude Simple Pendulum

Suppose we seek the period $T$ of a simple pendulum of length $l$ and mass $m$ as shown in Fig. 13.2, but we do not wish to make the conventional small amplitude approximation, under which the period $T_0$ is given by $2\pi \sqrt{l/g}$, where $g$ is the acceleration of gravity. With $y$ standing for the vertical coordinate of the pendulum (measured upward from its point of support), $I = ml^2$ for its moment of inertia about the point of support, and $\omega$ for its angular velocity, we start by noting that the total energy (kinetic plus potential) of the pendulum in a general position is given by

$$\text{energy} = \frac{1}{2} I \omega^2 + mgy = \frac{1}{2} ml^2 \left( \frac{d\theta}{dt} \right)^2 - mgl \cos \theta$$

(13.17)

If, with $\theta_0$ symbolizing the amplitude of the motion, we invoke conservation of energy, equating the energy at a general point to the energy at the highest point in the swing (where $d\theta/dt = 0$ and $\theta = \theta_0$), we find that

$$\frac{1}{2} ml^2 \left( \frac{d\theta}{dt} \right)^2 - mgl \cos \theta = -mgl \cos \theta_0 \quad \Rightarrow \quad \frac{d\theta}{dt} = \pm \sqrt{\frac{2g}{l}} \sqrt{\cos \theta - \cos \theta_0}$$

(13.18)

(We assume that the pendulum does not have sufficient energy to swing over the top.) This last relationship then leads to the conclusion that

$$dt = \pm \sqrt{\frac{l}{2g}} \frac{d\theta}{\sqrt{\cos \theta - \cos \theta_0}} \quad \Rightarrow \quad \int_0^{T/4} dt = \frac{T}{4} = \sqrt{\frac{l}{2g}} \int_0^{\theta_0} \frac{d\theta}{\sqrt{\cos \theta - \cos \theta_0}}$$

(13.19)
13.1. SAMPLE PROBLEMS

where we have integrated over one-quarter of the period ($0 < t < T/4$) in time and over one-quarter of the swing ($0 < \theta < \theta_0$) in angle, and we have taken the positive sign because, in its motion over this interval, the pendulum indeed has positive angular velocity. Further, we have assumed from symmetry that the full period $T$ is four times the time required for the pendulum to swing from its lowest point to its highest point.

This integral can be recast in numerous ways. Anticipating an existing standard function, we invoke the half angle identity $\cos \theta = 1 - 2\sin^2(\theta/2)$, finding that

$$T = 2\sqrt{\frac{l}{g}} \int_0^\theta \frac{d\theta}{\sqrt{\sin^2(\theta_0/2) - \sin^2(\theta/2)}}$$

(13.20)

We then introduce the variable $\phi$ defined by $\sin(\theta/2) = \sin(\theta_0/2) \sin \phi$ to find that

$$T(k) = 4\sqrt{\frac{l}{g}} \int_0^{\pi/2} \frac{d\phi}{\sqrt{1 - k^2 \sin^2 \phi}}$$

(13.21)

where the (dimensionless) parameter $k = \sin(\theta_0/2)$ is determined by the amplitude of the pendulum’s motion. By writing $T(k)$, we have drawn attention in the notation to the dependence of the period on $k$ (and hence on the amplitude). Note also that, when the amplitude is small, $k \approx 0$ and the integral can be readily evaluated to yield that $T_0 = 2\pi \sqrt{l/g}$, which—reassuringly—agrees with the known value quoted at the beginning of this subsection. Then, expressing the period $T(k)$ in units of $T_0$, we find finally that

$$\frac{T(k)}{T_0} = \frac{2}{\pi} \int_0^{\pi/2} \frac{d\phi}{\sqrt{1 - k^2 \sin^2 \phi}} = \frac{2}{\pi} K(k)$$

(13.22)

Here, for purposes of notation alone, we have introduced the integral

$$K(k) = \int_0^{\pi/2} \frac{d\phi}{\sqrt{1 - k^2 \sin^2 \phi}}$$

(13.23)

which defines a standard, tabulated function known as the complete elliptic integral of the first kind. Note that the value of this integral depends on a parameter not in a limit but in the very structure of the integrand. This integral falls into the third of our three categories.

13.1.5 Statistical Data Analysis

When repeated measurements of a single quantity are subject to a large number of individually small, random fluctuations, the distribution of those measurements about their mean follows the normal or Gaussian distribution function given by

$$G(x) = \frac{1}{\sqrt{2\pi}\sigma^2} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

(13.24)

where $\mu$ and $\sigma$ are the mean and standard deviation of the distribution, respectively, and $G(x)$ is normalized so that $\int_{-\infty}^{\infty} G(x) \, dx = 1$. The probability that a single measurement will lie between $\mu - \alpha \sigma$ and $\mu + \alpha \sigma$ is then given by

$$P(\mu - \alpha \sigma < x < \mu + \alpha \sigma) = \int_{\mu - \alpha \sigma}^{\mu + \alpha \sigma} G(x) \, dx = \frac{2}{\sqrt{\pi}} \int_0^{\alpha/\sqrt{2}} e^{-s^2} \, ds = \text{erf} \left( \frac{\alpha}{\sqrt{2}} \right)$$

(13.25)

where $s = (x - \mu)/(\sigma \sqrt{2})$ and, for purposes of notation alone, we have introduced the integral

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-s^2} \, ds$$

(13.26)
which defines a standard, tabulated function known as the error function. Thus, determining the probability given by Eq. (13.25) boils down to evaluating an integral that depends on a parameter appearing in the upper limit, i.e., to evaluating an integral that falls into the second of our three categories.

### 13.1.6 The Cornu Spiral

The two integrals

\[
C(u) = \int_0^u \cos \left( \frac{\pi t^2}{2} \right) \, dt \quad ; \quad S(u) = \int_0^u \sin \left( \frac{\pi t^2}{2} \right) \, dt
\]

which are called the Fresnel integrals, appear in the study of Fresnel diffraction. Together, they define the Cornu spiral, which is a graph of \( S(u) \) versus \( C(u) \). Each is a function of its upper limit as a parameter and falls into the second of our three categories.

### 13.1.7 Electric and Magnetic Fields and Potentials

Among the richest sources of important—and frequently analytically intractable—integrals are the relationships in electromagnetic theory that determine fields and potentials from prescribed sources. For a distribution of static charge, for example, we identify an element of that charge and, in mks units, find the electric field \( E(r) \) and the electrostatic potential \( V(r) \) at the point \( r \) by evaluating the integrals

\[
E(r) = \frac{1}{4\pi\epsilon_0} \int \frac{\mathbf{r} - \mathbf{r}'}{|\mathbf{r} - \mathbf{r}'|^3} \, dq' \quad \text{and} \quad V(r) = \frac{1}{4\pi\epsilon_0} \int \frac{dq'}{|\mathbf{r} - \mathbf{r}'|}
\]

which extend over all charges in the source. Similarly, for a source consisting of a steady current \( I' \) in a wire, we identify an element \( dr' \) of the wire located at \( r' \) and, in mks units, find the magnetic field \( B(r) \) and the magnetic vector potential \( A(r) \) at the point \( r \) by evaluating the integrals

\[
B(r) = \frac{\mu_0}{4\pi} \int \frac{I' \, dr' \times (\mathbf{r} - \mathbf{r}')}{|\mathbf{r} - \mathbf{r}'|^3} \quad \text{and} \quad A(r) = \frac{\mu_0}{4\pi} \int \frac{I' \, dr'}{|\mathbf{r} - \mathbf{r}'|}
\]

which extend over the path followed by the current.

More specifically, suppose we seek the electrostatic potential in the plane midway between two identical uniformly charged circular rings of radius \( a \) with their planes parallel, their axes coincident, and their centers separated by \( 2a \). The envisioned situation is shown in Fig. 13.3. Each ring carries a total charge \( Q \) with (linear) charge density \( \lambda \). Using cylindrical coordinates \((r, \phi, z)\), we first find the potential at the general point \( r = r \cos \phi \, \mathbf{i} + r \sin \phi \, \mathbf{j} + z \, \mathbf{k} \), produced by one such ring positioned in the \( xy \) plane with its center at the origin. Let \( r' = a \cos \phi' \, \mathbf{i} + a \sin \phi' \, \mathbf{j} \) locate an element on that (single) ring. In this notation,

\[
\mathbf{r} - \mathbf{r}' = (a \cos \phi - a \cos \phi') \, \mathbf{i} + (a \sin \phi - a \sin \phi') \, \mathbf{j} + z \, \mathbf{k}
\]

so

\[
|\mathbf{r} - \mathbf{r}'| = \left[ (r \cos \phi - a \cos \phi')^2 + (r \sin \phi - a \sin \phi')^2 + z^2 \right]^{1/2} = \left[ r^2 + a^2 - 2ar \cos(\phi' - \phi) + z^2 \right]^{1/2}
\]

Placing these results into the second member of Eq. (13.28) and recognizing that \( dq' = \lambda a \, d\phi' \), we at last find that

\[
V_{\text{one}}(r, \phi, z) = \frac{1}{4\pi\epsilon_0} \int_0^{2\pi} \frac{\lambda a \, d\phi'}{|r^2 + a^2 - 2ar \cos(\phi' - \phi) + z^2|^{1/2}}
\]
13.1. SAMPLE PROBLEMS

Figure 13.3: The potential produced by two charged rings. Part (a) shows the geometry of ultimate interest. Part (b) shows the simple situation with one ring used as a stepping stone.

(a)

(b)

The entire integral assumes a simpler appearance, however, on the variable $\alpha = \phi' - \phi$, becoming

$$V_{\text{one}}(r, \phi, z) = \frac{\lambda a}{4 \pi \epsilon_0} \int_0^{2\pi} d\alpha \frac{\left[r^2 + a^2 - 2ar \cos \alpha + z^2\right]^{1/2}}{[r^2 + 2a^2 - 2ar \cos \alpha]^{1/2}}$$  \hspace{1cm} (13.33)

where we have invoked the periodicity of the integrand in $\alpha$ to justify writing the integral to run from 0 to $2\pi$ rather than from $-\phi$ to $2\pi - \phi$. (The integral extends over an entire period of the integrand in either case.) As implied by the symmetry, the potential has turned out not to depend on $\phi$.

We can now return to our original question, which asked about the potential in the midplane when two rings of the sort to which Eq. (13.33) applies have their centers separated by $2a$. The observation point in the midplane is thus a distance $a$ “above” one of the rings and the same distance $a$ “below” the other. We find the potential produced by these two rings by adding a contribution from each ring, concluding that

$$V_{\text{midplane}}(r) = V_{\text{one}}(r, \phi, -a) + V_{\text{one}}(r, \phi, a) = \frac{2\lambda a}{4 \pi \epsilon_0} \int_0^{2\pi} d\alpha \frac{s^2 - 2s \cos \alpha + 2}{[s^2 - 2s \cos \alpha + 2]^{1/2}}$$  \hspace{1cm} (13.34)

Finally, introducing the variable $s$ defined by $r = sa$ to express the radial coordinate in dimensionless terms, we find the expression

$$V_{\text{two}}(s) = V_{\text{midplane}}(sa) = \frac{2\lambda a}{4 \pi \epsilon_0 a} \int_0^{2\pi} d\alpha \frac{d\alpha}{[s^2 - 2s \cos \alpha + 2]^{1/2}}$$  \hspace{1cm} (13.35)

or, even better, the expression

$$V(s) = \frac{\sqrt{2}}{2\pi} \int_0^{2\pi} d\alpha \frac{d\alpha}{[s^2 - 2s \cos \alpha + 2]^{1/2}}$$  \hspace{1cm} (13.36)

where $V(s) = V_{\text{two}}(s)/V_{\text{two}}(0)$. Equation (13.36) gives the potential at the radial coordinate $r = sa$ in the midplane between two uniformly charged rings of radius $a$ separated by $2a$. This integral falls into the third of our three categories, since its value is a function of the parameter $s$ in the integrand.
13.1.8 Quantum Probabilities

In the standard interpretation, the wave function \( \psi(x) \) for a one-dimensional quantum system is a probability amplitude, and the quantity \( |\psi(x)|^2 \) is a probability density. Further the wave function by convention is normalized so that \( \int_{-\infty}^{\infty} |\psi(x)|^2 \, dx = 1 \). Thus, the integral

\[
P(x_1 < x < x_2) = \int_{x_1}^{x_2} |\psi(x)|^2 \, dx \tag{13.37}
\]
gives the probability of finding the quantum system with its coordinate somewhere between \( x = x_1 \) and \( x = x_2 \).

More specifically, we remember that the wave function for a quantum harmonic oscillator in its ground state is given by

\[
\psi(x) = \left( \frac{m \omega}{\pi \hbar} \right)^{1/4} e^{-y^2/2} \tag{13.38}
\]

where \( \hbar \) is Planck’s constant, \( m \) is the mass of the oscillator and, with \( k \) the spring constant and \( a = \sqrt{\hbar \omega / k} \), \( \omega = \sqrt{k/m} \) is the frequency of the oscillator; \( y = x/a \) is a dimensionless coordinate.

Since the classical turning point of the oscillator occurs when its energy (\( \frac{1}{2} \hbar \omega \) for the ground state) is equal to the potential energy \( \frac{1}{2} k x_{\text{turn}}^2 \), the turning point of this oscillator is given by

\[
\frac{1}{2} \hbar \omega = \frac{1}{2} k x_{\text{turn}}^2 \quad \Rightarrow \quad x_{\text{turn}} = \sqrt{\frac{\hbar \omega}{k}} = a \tag{13.39}
\]

which provides a classical interpretation for the parameter \( a \). The probability that the particle in the ground state of a quantum harmonic oscillator will be found in the classically forbidden region (i.e., somewhere outside the classical turning point) is given by the integral

\[
P(|x| > |x_{\text{turn}}|) = \int_{-\infty}^{-a} |\psi(x)|^2 \, dx + \int_{a}^{\infty} |\psi(x)|^2 \, dx = 1 - \int_{-a}^{a} |\psi(x)|^2 \, dx \tag{13.40}
\]

Finally, after substituting the wave function and recasting the entire integral as an integral on the variable \( y \), we find that

\[
P(|x| > |x_{\text{turn}}|) = 1 - \frac{1}{\sqrt{\pi}} \int_{-1}^{1} e^{-y^2} \, dy = 1 - \frac{2}{\sqrt{\pi}} \int_{0}^{1} e^{-y^2} \, dy = 1 - \text{erf}(1) \tag{13.41}
\]

which contains no parameters and thus falls into the first of our three categories.

13.1.9 Expansion in Orthogonal Functions

Suppose we have identified a set of functions \( \phi_i(x) \), \( i = 1, 2, 3, \ldots \), that have the property

\[
\int_{a}^{b} \phi_i(x) \phi_j(x) \, w(x) \, dx = N_j \delta_{ij} \tag{13.42}
\]

where \( N_j \) is a constant, \( \delta_{ij} \) is the Kronecker delta (which has the value 1 when the indices are equal and the value 0 otherwise), \( w(x) \) is a known weight function, and \( a \) and \( b \) define a known interval. These functions are said to be orthogonal with weight \( w(x) \) on the interval \( a \leq x \leq b \). The members of this set provide a basis in terms of which any arbitrary function \( f(x) \) defined on the same interval can be expanded in the form

\[
f(x) = \sum_{n} a_n \phi_n(x) \tag{13.43}
\]
13.5. ALGORITHMS FOR NUMERICAL INTEGRATION

119

Though the argument we will here present is not mathematically rigorous, the expansion coefficients $a_n$ can be quickly determined by multiplying Eq. (13.43) by $w(x) \phi_j(x)$, integrating over the interval $a \leq x \leq b$, and exchanging the order of integration and summation to find that

$$\int_a^b w(x) \phi_j(x) f(x) dx = \sum_n a_n \int_a^b w(x) \phi_j(x) \phi_n(x) dx = \sum_n a_n N_n \delta_{nj} = N_j a_j \quad (13.44)$$

We conclude that, if we know either of $f(x)$ or $a_n$, we can determine the other, i.e., that

$$f(x) = \sum_n a_n \phi_n(x) \quad \iff \quad a_n = \frac{1}{N_n} \int_a^b w(x) \phi_n(x) f(x) dx \quad (13.45)$$

The determination of the coefficients in this expansion of a known function clearly involves the evaluation of integrals, which explains the inclusion of this example in this chapter.

While many sets of orthogonal functions could be enumerated (see exercises), probably the most common set is

$$\{ 1, \sin \frac{n \pi x}{l}, \cos \frac{m \pi x}{l}; \quad n, m = 1, 2, 3, \ldots \} \quad (13.46)$$

Direct evaluation of the integral of each member of this set with all other members will reveal that these functions are orthogonal on the interval $-l \leq x \leq l$ with weight $w(x) = 1$, i.e., that

$$\int_{-l}^l 1 \times 1 dx = 2l \quad ; \quad \int_{-l}^l 1 \times \sin \frac{n \pi x}{l} dx = 0 \quad ; \quad \int_{-l}^l 1 \times \cos \frac{n \pi x}{l} dx = 0 \quad (13.47)$$

$$\int_{-l}^l \sin \frac{n \pi x}{l} \sin \frac{m \pi x}{l} dx = l \delta_{nm} \quad ; \quad \int_{-l}^l \sin \frac{n \pi x}{l} \cos \frac{m \pi x}{l} dx = 0 \quad (13.48)$$

$$\int_{-l}^l \cos \frac{n \pi x}{l} \cos \frac{m \pi x}{l} dx = l \delta_{nm} \quad ; \quad \int_{-l}^l \cos \frac{n \pi x}{l} \sin \frac{m \pi x}{l} dx = 0 \quad (13.49)$$

The expansion

$$f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty} \left( a_n \cos \frac{n \pi x}{l} + b_n \sin \frac{n \pi x}{l} \right) \quad (13.50)$$

of a function $f(x)$ that is periodic with period $2l$ in this set of orthogonal functions is called a Fourier series. The coefficients are given by the integrals

$$a_n = \frac{1}{l} \int_{-l}^l f(x) \cos \frac{n \pi x}{l} dx \quad ; \quad b_n = \frac{1}{l} \int_{-l}^l f(x) \sin \frac{n \pi x}{l} dx \quad (13.51)$$

13.5 Algorithms for Numerical Integration

Unfortunately, very many integrals of great interest have no closed form, analytic evaluation. To address these integrals, numerical analysts have developed many formulae—often called quadrature formulæ—for numerical integration. In this section, we describe several of these formulæ.

13.5.1 Newton-Cotes Quadrature

One family of quadrature formulæ can be deduced by starting with the recognition that the definite integral

$$A = \int_a^b f(x) dx \quad (13.52)$$

$^1$To put this set into the form of the previous paragraph, we would identify $\phi_1(x) = 1$, $\phi_2(x) = \sin(\pi x/l)$, $\phi_3(x) = \cos(\pi x/l)$, $\phi_4(x) = \sin(2\pi x/l)$, ...$^2$The first coefficient is written $a_0/2$ rather than $a_0$ to simplify the integrals determining the coefficients. With the choice we have made, the integral for $a_0$ turns out to be obtainable by setting $n = 0$ in the expression for $a_n$; we do not need a third—and different—expression for that one coefficient.
Figure 13.4: Division of interval \( a < x < b \) into \( N \) segments.

\[
x_0 \quad x_1 \quad x_2 \quad x_3 \quad x_4 \quad \cdots \quad x_{N-3} \quad x_{N-2} \quad x_{N-1} \quad x_N
\]

\( a \quad x \quad 0 \quad x \quad 1 \quad x \quad 2 \quad x \quad 3 \quad x \quad 4 \quad x \quad N \quad - \quad 3 \quad x \quad N \quad - \quad 2 \quad x \quad N \quad - \quad 1 \quad x \quad N
\]

Figure 13.5: Two different approximations leading to quadrature formulae. The left figure corresponds to Eq. (13.53), the right figure to Eq. (13.55).

\( (a) \quad (b) \)

represents geometrically the area under the graph of \( f(x) \) over the interval \( a \leq x \leq b \). As shown in Fig. 13.4, let that interval be divided into \( N \) segments, each of width \( \Delta x = (b - a)/N \), let \( x_0 = a \), \( x_1 = a + \Delta x \), \( x_2 = a + 2 \Delta x \), \( \ldots \), \( x_i = a + i \Delta x \), \( \ldots \), \( x_N = b \), and let \( f(x_0) = f(a) = f_0 \), \( f(x_1) = f_1 \), \( \ldots \), \( f(x_i) = f_i \), \( \ldots \), \( f(x_N) = f(b) = f_N \). To deduce the simplest quadrature formula, we approximate the area of each resulting strip by the area of a rectangle whose height is the value of \( f(x) \) at the left end of the strip [Fig. 13.5(a)]. Thus

\[
\int_a^b f(x) \, dx \approx f_0 \Delta x + f_1 \Delta x + \cdots + f_i \Delta x + \cdots + f_{N-1} \Delta x
\]

which turns out to be 100% accurate only if \( f(x) \) happens to be a constant.

If, however, we approximate the area of each strip by the area of a rectangle whose height is the value of \( f(x) \) at the midpoint of the strip, we would deduce the midpoint rule

\[
\int_a^b f(x) \, dx \approx M_N = \left( f_{1/2} + f_{3/2} + f_{5/2} + \cdots + f_{N-3/2} + f_{N-1/2} \right) \Delta x
\]

which turns out to be 100% accurate when \( f(x) \) is a linear function of \( x \). (When \( f(x) \) is linear, the error made by overestimating the function in one half of the interval is exactly compensated by the error made by underestimating the function in the other half of the interval. The formula turns out to be 100% accurate for a polynomial of one higher order than the polynomial used—here a constant—to approximate the function in each strip.)

For a further refinement, we might approximate the area of each strip by the area of a trapezoid [Fig. 13.5(b)], in which case we obtain the trapezoidal rule,

\[
\int_a^b f(x) \, dx \approx T_N = \frac{1}{2}(f_0 + f_1) \Delta x + \frac{1}{2}(f_1 + f_2) \Delta x + \cdots + \frac{1}{2}(f_{N-1} + f_N) \Delta x
\]
13.5. ALGORITHMS FOR NUMERICAL INTEGRATION

Figure 13.6: A rearrangement of the labels for dividing points.

\[ a \quad x_4 \quad x_2 \quad x_1 \quad x_6 \quad x_3 \quad x_7 \quad b \]

\[
\begin{align*}
\int_a^b f(x) \, dx &\approx S_N = \frac{1}{3} (f_0 + 4f_1 + 2f_2 + \cdots + 4f_{N-2} + 2f_{N-1} + f_N) \, \Delta x
\end{align*}
\]

which, as with the midpoint rule, is 100% accurate when \( f(x) \) is linear, i.e., when \( f(x) \) is a polynomial of the same order as the one used to approximate the function.

A final (for here) and still better approximation is obtained if we pair the strips—which then requires \( N \) to be even—and approximate the area of each pair by the area under the parabola fitted to the values of \( f(x) \) at the three points defining the pair. The result,

\[
\int_a^b f(x) \, dx \approx S_N = \frac{1}{3} (f_0 + 4f_1 + 2f_2 + \cdots + 2f_{N-2} + 4f_{N-1} + f_N) \, \Delta x
\]  

is called Simpson’s rule. (See the first exercise in Section 13.15.2.) For reasons similar to those that apply for the midpoint rule, Simpson’s rule is 100% accurate for cubic polynomials, one order higher than the quadratic polynomial used to approximate the function.

Continuation of this procedure to higher and higher degree polynomials generates a succession of increasingly more accurate—but also more and more complicated—Newton-Cotes formulae, which are characterized in particular by evaluating the function at equally spaced interpolation points.

13.5.2 Rearrangements for Computational Efficiency

Two rearrangements of the formulae deduced to this point facilitate the writing of more efficient algorithms. Suppose, for example, that we use the trapezoidal rule and write out a succession of formulae for evaluating \( \int_a^b f(x) \, dx \) for \( N = 1, 2, 4, 8, 16, \ldots \) divisions of the interval of integration. We find first that

\[
T_1 = T_{2^0} = \frac{f(a) + f(b)}{2} (b - a)
\]

Then, halving the step size and introducing \( x_1 = a + (b - a)/2 \), the midpoint of the interval (see Fig. 13.6, in which—for present convenience—we label the points differently than we did in Fig. 13.4), we find that

\[
T_2 = T_{2^1} = \left[ \frac{f(a)}{2} + f(x_1) + \frac{f(b)}{2} \right] \frac{(b - a)}{2} = \frac{1}{2} T_1 + f(x_1) \frac{(b - a)}{2}
\]

\[
= \frac{1}{2} T_1 + f \left( a + \frac{b - a}{2} \right) \frac{(b - a)}{2}
\]  

(13.58)

Halving the step size again and introducing \( x_2 = a + (b - a)/4 \) and \( x_3 = a + 3(b - a)/4 \) (again, see Fig. 13.6), we find that

\[
T_4 = T_{2^2} = \left[ \frac{f(a)}{2} + f(x_2) + f(x_1) + f(x_3) + \frac{f(b)}{2} \right] \frac{(b - a)}{4}
\]
Fig. 13.6, we find, for example, that extrapolation and, in general, with

\[
T_4 = \frac{1}{2} T_2 + \left[ f(x_2) + f(x_3) \right] \frac{(b-a)}{4}
\]

and continuing one more step, we find that

\[
T_8 = \frac{1}{2} T_4 + \left[ f(x_4) + f(x_5) + f(x_6) + f(x_7) \right] \frac{(b-a)}{8}
\]

Yet again, introducing

\[
x_4 = a + \frac{1}{8}(b-a) \quad x_5 = a + \frac{3}{8}(b-a) \\
x_6 = a + \frac{5}{8}(b-a) \quad x_7 = a + \frac{7}{8}(b-a)
\]

and, in general, with

\[
x_{2N+1} = a + \frac{(2j+1)(b-a)}{2^{N+1}} \quad 0 \leq j < 2^{N+1}
\]

that

\[
T_{2N+1} = \frac{1}{2} T_{2N} + \left[ f \left( a + \frac{b-a}{2^{N+1}} \right) + f \left( a + \frac{3b-a}{2^{N+1}} \right) \right. \\
\left. + f \left( a + \frac{5b-a}{2^{N+1}} \right) \right] \frac{(b-a)}{2^{N+1}}
\]

Evidently, we can step from any evaluation by the trapezoidal rule to an evaluation by the trapezoidal rule with twice as many divisions without recalculating anything that we have already calculated! We shall refer to this embellishment as the recursive trapezoidal rule.

A second strategy for making algorithms more efficient involves what is called Richardson extrapolation. Using the trapezoidal rule of Eq. (13.55) and labeling the interpolation points as in Fig. 13.6, we find, for example, that

\[
T_4 = \left[ \frac{1}{2} f(a) + f(x_2) + f(x_1) + f(x_3) + \frac{1}{2} f(b) \right] \frac{b-a}{4}
\]

and that

\[
T_8 = \left[ \frac{1}{2} f(a) + f(x_4) + f(x_2) + f(x_5) \right. \\
\left. + f(x_1) + f(x_6) + f(x_3) + f(x_7) + \frac{1}{2} f(b) \right] \frac{b-a}{8}
\]

Note, in particular, the combination

\[
4T_8 - T_4 = \left[ f(a) + 4f(x_4) + 2f(x_2) + 4f(x_5) + 2f(x_1) \right. \\
\left. + 4f(x_6) + 2f(x_3) + 4f(x_7) + f(b) \right] \left( \frac{b-a}{8 \times 3} \right)
\]
and recognize that \((b - a)/8 = \Delta x\) is the width of a single strip when the interval \(a < x < b\) is divided into eight segments. Thus, we can write this last expression as

\[
\frac{4T_8 - T_4}{3} = \left[ f(a) + 4f(x_4) + 2f(x_5) + 4f(x_7) + 2f(x_1) + 4f(x_6) + 2f(x_3) + 4f(x_7) + f(b) \right] \left( \frac{\Delta x}{3} \right)
\]

(13.67)

which—mirabile dictu—we recognize as Simpson’s rule for evaluating the integral with 8 divisions of the interval! We conclude that

\[
S_8 = \frac{4T_8 - T_4}{3}
\]

(13.68)

More generally, we could also conclude that

\[
S_{2n} = \frac{4T_{2n} - T_n}{3}
\]

(13.69)

The extrapolation formula of Eq. (13.69) applied to two successive evaluations by the trapezoidal rule gives the result of evaluation by Simpson’s rule!

A subroutine for trapezoidal integration can thus be used in a very efficient algorithm for evaluating integrals by Simpson’s rule. We evaluate the integral twice by the trapezoidal rule for two values of \(n\), one of which is twice the other, finding \(T_n\) and \(T_{2n}\). Then we find the Simpson’s rule evaluation by exploiting the extrapolation formula of Eq. (13.69), which expresses the first step in what is called Romberg integration. We do actual numerical integration only with the trapezoidal rule, and we invoke the efficiency described in the first paragraph of this subsection in doing that. A routine for integration via the trapezoidal rule can thus be the workhorse for many other routines.

Indeed, Romberg integration goes beyond simply generating evaluations by Simpson’s rule from evaluations by the trapezoidal rule. Suppose we used the trapezoidal rule to generate the four values \(T_n, T_{2n}, T_{4n}\), and \(T_{8n}\). We could then use Eq. (13.69) to generate the three values \(S_{2n}, S_{4n}\), and \(S_{8n}\). In Romberg integration, we next generate a pair of still more accurate values from the formulæ \(X_{4n} = (16S_{4n} - S_{2n})/15\) and \(X_{8n} = (16S_{8n} - S_{4n})/15\); and then we generate a further improved value from the formula \(Y_{8n} = (64X_{8n} - X_{4n})/63\). This process can, of course, be continued indefinitely—though we rarely have to go even as far as we have described.

### 13.5.3 Assessing Error

Numerical evaluations, of course, only approximate the integral. Two distinctly different sorts of errors can occur. Truncation errors arise because the integral has been approximated by a finite sum; roundoff errors arise because computers do not store non-integers to 100\% precision and, in the evaluation of a sum, the roundoff error can accumulate as the number of arithmetic operations increases. Truncation errors become smaller as the width of strips is reduced. Roundoff errors, unfortunately, become more significant as the width of strips is reduced (because, with narrower strips, more arithmetic must be done). Usually, roundoff errors are negligible, the more so as the sophistication of the algorithm increases (and, hence, the amount of arithmetic decreases). Provided we do not strive for accuracy greater than about 1 part in \(10^5\) or \(10^6\) (with single precision floating point arithmetic), we can usually ignore roundoff errors. Thus, provided the function being integrated is such that the algorithm converges fairly rapidly with decreasing strip width, the quickest way to obtain a reasonably reliable estimate of truncation error is to evaluate the integral with two different step widths, the second being half of the first, and compare the two results. Presuming that roundoff error has not begun to be important, we can be confident that the second result is more accurate than the first. Thus, if the two agree to 1 part in \(10^5\), say, we can with reasonable confidence, assume that the second value is good to one part in \(10^5\). Indeed, the second value is probably better than that, but assessing its accuracy by this method would entail obtaining a third value by using a strip width half of that used to determine
the second value.\textsuperscript{3} Indeed, one strategy for achieving a desired accuracy with reasonable certainty is to evaluate an integral repeatedly by a particular method, halving the strip width each time, and continuing until the new value received differs from its predecessor by less than the desired accuracy (though we must be careful not to push this approach so far that roundoff problems within the computer begin to become significant).\textsuperscript{4}

From a more sophisticated perspective, numerical analysts have deduced expressions for the error in various Newton-Cotes formulae.\textsuperscript{5} For the midpoint formula of Eq. (13.54), for example,

$$\left| \int_a^b f(x) \, dx - M_N \right| = \frac{(b-a)^3}{24N^2} \left| \frac{d^2 f}{dx^2} \right|_{x=\xi}$$  \hspace{1cm} (13.70)

where $\xi$ is some value of $x$ satisfying $a < \xi < b$—an expression that is valid provided the function $f(x)$ satisfies suitable requirements on continuity. The similar expressions

$$\left| \int_a^b f(x) \, dx - T_N \right| = \frac{(b-a)^3}{12N^2} \left| \frac{d^2 f}{dx^2} \right|_{x=\xi}$$  \hspace{1cm} (13.71)

and

$$\left| \int_a^b f(x) \, dx - S_N \right| = \frac{(b-a)^5}{180N^4} \left| \frac{d^4 f}{dx^4} \right|_{x=\xi}$$  \hspace{1cm} (13.72)

can be derived for the trapezoidal rule given by Eq. (13.55) and for Simpson’s rule given by Eq. (13.56), respectively. Again, $\xi$ is a value somewhere between $x = a$ and $x = b$, though it is not likely to have the same value in all three formulae.

These results do not, of course, tell us how to determine the error exactly because they don’t tell us how to determine $\xi$ exactly. Even so, they are not entirely useless, having at least two particular values:

1. If it should happen in the first two cases that $d^2 f/dx^2 = 0$ or in the third case that $d^4 f/dx^4 = 0$ \textit{throughout} the interval of integration, then the error is zero, since the right hand side of these expressions gives zero for all possible values of $\xi$. Thus, these formulae confirm our previous assertions that the midpoint and trapezoidal rules will be 100\% accurate for linear functions and that Simpson’s rule will be 100\% accurate for cubic polynomials.

2. If $D_{\text{max}}(i)$ is the \textit{maximum} value of $d^i f/dx^i$ in the interval $a < x < b$, then the above expressions support the inequalities

$$\left| \int_a^b f(x) \, dx - M_N \right| \leq \frac{(b-a)^3}{24N^2} D_{\text{max}}(2)$$  \hspace{1cm} (13.73)

$$\left| \int_a^b f(x) \, dx - T_N \right| \leq \frac{(b-a)^3}{12N^2} D_{\text{max}}(2)$$  \hspace{1cm} (13.74)

$$\left| \int_a^b f(x) \, dx - S_N \right| \leq \frac{(b-a)^5}{180N^4} D_{\text{max}}(4)$$  \hspace{1cm} (13.75)

(though we must keep in mind that these approximations will frequently be extremely crude, so these upper bounds may well be very conservative). Provided that problems with computer

\textsuperscript{3}We shall make this criterion a bit more explicit in the next paragraphs.

\textsuperscript{4}We shall see in later sections how we might decide when roundoff has started to be significant.

\textsuperscript{5}These formulae may be found in several sources, including Philip J. Davis and Philip Rabinowitz, \textit{Numerical Integration} (Blaisdell Publishing Company, Waltham, MA, 1967), pp. 17, 20; and Milton Abramowitz and Irene A. Stegun, \textit{Handbook of Mathematical Functions} (National Bureau of Standards Applied Mathematics Series 55, June, 1964), pp. 885ff.
roundoff do not begin to appear, we conclude from these results that an upper bound on the error in the midpoint and trapezoidal rules falls off like \(1/N^2\) while that bound in Simpson’s rule falls off like \(1/N^4\). Doubling \(N\) therefore reduces the error in the midpoint and trapezoidal rules by a factor of four, while doubling \(N\) reduces the error in Simpson’s rule by a factor of sixteen. With Simpson’s rule, every doubling of \(N\) should gain at least one more decimal point in accuracy, so the convergence criterion described in the first paragraph in this subsection is particularly apt when Simpson’s rule is used.

13.5.4 Iterative and Adaptive Algorithms

In the previous subsections, we assumed that the user of a particular algorithm would actually view the value obtained for a succession of values of \(N\) and decide personally when to stop by examining the changes that occur as \(N\) is successively doubled. We can, of course, program a computer to make those decisions. One extremely common approach exploits Simpson’s rule (probably via the trapezoidal rule and Romberg integration) to obtain \(S_2\), \(S_4\), \(S_8\), ..., compares each new value with its predecessor and stops when the absolute value of the difference is smaller than a tolerance—either absolute or relative—prescribed in advance. As a guard against an infinite loop, these algorithms should also stop if the desired tolerance has not been achieved in some maximum number of refinements and should print a warning when the desired tolerance has not been achieved. This method is said to be iterative, because it generates a succession of results, examines each new result in turn, and repeats the process until the new result meets or exceeds the prescribed tolerance. The points at which the function is evaluated, however, are determined ahead of time and are not influenced at all by the nature of the specific integrand to which the algorithm is applied.

Another family of algorithms (which may be iterative or noniterative) aims to minimize computational labor by estimating—though the methods for doing so are often crude—the accuracy obtained with each strip as the evaluation unfolds and shrinking or enlarging that strip to achieve a particular tolerance before going on to the next strip. In these adaptive methods, the points at which the function is evaluated are adjusted in response to the particular function being integrated. Because the assessment of accuracy at a particular strip can result either in shrinking or enlarging the width of that—or the next—strip, adaptive methods focus the computational effort in regions where the function varies rapidly and give less attention to regions in which the function varies slowly.

13.5.5 Gaussian Quadrature

The approach of Gaussian quadrature to numerical integration is more complicated than the Newton-Cotes approach but significantly better in some respects. In the Gaussian approach, both the points at which the function is to be evaluated and the weights to be applied to each value are adjusted to achieve maximum accuracy when the function is approximated by a polynomial of a given order.

The development of a formula for Gaussian quadrature is simplified if we begin by introducing a set of \(m + 1\) points \(t_i\), \((i = 0, 1, 2, \ldots, m)\) that divide the interval \(t_0 = a \leq t \leq b = t_m\) into \(m\) segments, the \(i\)-th of which extends over the interval \(t_{i-1} \leq t \leq t_i\). The values \(t_i\) may—but need not—be equally spaced. In this notation, we write the integral of interest as a sum of integrals over each segment, i.e., we write

\[
\int_a^b g(t) \, dt = \sum_{i=1}^{m} \int_{t_i}^{t_{i+1}} g(t) \, dt
\]  

To facilitate the discussion, however, we rescale and translate the variable in the \(i\)-th segment by introducing the variable \(x\) defined by

\[
x = \frac{2t - (t_{i+1} + t_i)}{t_{i+1} - t_i} \quad \text{or} \quad t = \frac{t_{i+1} + t_i}{2} + \frac{t_{i+1} - t_i}{2} x = t_i^{\text{mid}} + \frac{\Delta t_i}{2} x
\]  

(13.77)
where \( t_i^{\text{mid}} \) is the coordinate at the midpoint of the \( i \)-th segment and \( \Delta t_i \) is the width of the \( i \)-th segment. With this change, \( x \) ranges from \(-1\) to \(+1\) as \( t \) ranges from \( t_i \) to \( t_{i+1} \), so the integrals of interest now assume the form

\[
\int_a^b g(t) \, dt = \sum_{i=1}^m \int_{t_i}^{t_{i+1}} g(t) \, dt = \sum_{i=1}^m \frac{\Delta t_i}{2} \int_{-1}^1 g \left( \frac{t_i^{\text{mid}} + \Delta t_i}{2} x \right) \, dx = \sum_{i=1}^m \frac{\Delta t_i}{2} \int_{-1}^1 f_i(x) \, dx \tag{13.78}
\]

where \( f_i(x) = g(t_i^{\text{mid}} + \Delta t_i x/2) \). In essence, then, we must evaluate an integral of the form

\[
\int_{-1}^1 f(x) \, dx \tag{13.79}
\]

where we omit the subscript \( i \) on \( f \) for the sake of a simpler notation. If we can find a useful numerical evaluation for the integral in this standard form, then all else can be obtained by appropriate translations and rescalings.

The strategy for Gaussian integration now involves selecting the number of points—say \( N \)—at which the function is to be evaluated in the interval \(-1 < x < 1\), assuming an approximate formula of the form

\[
\int_{-1}^1 f(x) \, dx = \sum_{k=1}^N w_k f(x_k) \tag{13.80}
\]

and then choosing both the weights \( w_k \) and the points of evaluation \( x_k \) to make this expression 100% accurate for a polynomial of as high an order as possible. Since we have \( 2N \) parameters to be determined, we should be able to make this expression accurate for a polynomial of order \( 2N - 1 \) with only \( N \) evaluations of the integrand.

To illustrate Gaussian integration more explicitly, let us derive a two-point formula, for which Eq. (13.80) would assume the more explicit form

\[
\int_{-1}^1 f(x) \, dx = w_1 f(x_1) + w_2 f(x_2) \tag{13.81}
\]

We choose \( w_1, w_2, x_1 \) and \( x_2 \) so that the formula gives the correct answer for the special cases

\[
\begin{align*}
  f(x) &= 1, \text{ yielding that} \quad \int_{-1}^1 dx = 2 = w_1 + w_2 \\
  f(x) &= x, \text{ yielding that} \quad \int_{-1}^1 x \, dx = 0 = w_1 x_1 + w_2 x_2 \\
  f(x) &= x^2, \text{ yielding that} \quad \int_{-1}^1 x^2 \, dx = \frac{2}{3} = w_1 x_1^2 + w_2 x_2^2 \\
  f(x) &= x^3, \text{ yielding that} \quad \int_{-1}^1 x^3 \, dx = 0 = w_1 x_1^3 + w_2 x_2^3
\end{align*}
\]

Because the integral is a linear function of its integrand, a formula that yields the correct answer in these four cases will also yield the correct answer for any linear combination of these special cases, i.e., for any cubic polynomial. These four equations determine the four unknowns. Eqs. (13.83) and (13.85) imply that

\[
  w_1 x_1 = -w_2 x_2 \quad \text{and} \quad w_1 x_1^3 = -w_2 x_2^3 \tag{13.86}
\]

which, when we divide the second by the first, yields \( x_1^2 = x_2^2 \), implying that \( x_1 = -x_2 \). (We reject the plus sign so the two values will be distinct.) Then Eqs. (13.84) and (13.82) yield that

\[
\frac{2}{3} = (w_1 + w_2) x_2^2 = 2x_2^2 \quad \Rightarrow \quad x_2^2 = \frac{1}{3} \tag{13.87}
\]
from which we also conclude that \( x_1^2 = \frac{1}{3} \). Thus,

\[
\begin{align*}
x_1 &= -\frac{1}{\sqrt{3}} & \text{and} & & x_2 &= \frac{1}{\sqrt{3}}
\end{align*}
\]  

(13.88)

Next, Eq. (13.83) implies that \( w_1 = w_2 \) and then Eq. (13.82) implies that \( w_1 = w_2 = 1 \). We conclude that

\[
\int_{-1}^{1} f(x) \, dx = f \left( -\frac{1}{\sqrt{3}} \right) + f \left( \frac{1}{\sqrt{3}} \right)
\]  

(13.89)

The result is exact for cubic polynomials with only two evaluations of the function per strip! Simpson’s rule, also exact for cubic polynomials, requires three evaluations of the function per strip.\(^6\)

Returning to the original function and variables as laid out in Eq. (13.78), we finally find that Eq. (13.89) supports the expression

\[
\int_{a}^{b} g(t) \, dt = \sum_{i=1}^{m} \frac{\Delta t_i}{2} \left[ g \left( t_i^{\text{mid}} - \frac{\Delta t_i}{2} \frac{1}{\sqrt{3}} \right) + g \left( t_i^{\text{mid}} + \frac{\Delta t_i}{2} \frac{1}{\sqrt{3}} \right) \right]
\]

(13.90)

The strategy invoked to develop the two-point Gaussian formula can also be applied to deduce higher order formulae. For a three-point formula, for example, we would have three points and three weights, and we would expect to be able to choose these unknowns to generate a formula that would be exact for a fifth-degree polynomial. The five-point formula

\[
\int_{-1}^{1} f(x) \, dx = 0.23692689 f(-0.90617985) + 0.47862867 f(-0.53846931) + 0.56888889 f(0.00000000) + 0.47862867 f(0.53846931) + 0.23692689 f(0.90617985)
\]

(13.91)

which is 100% accurate for polynomials of the ninth-degree or lower, is among the most popular of the formulae in this class. For the sake of later examples, we note that, in this expression

\[
\begin{align*}
w_1 &= 0.23692689 & x_1 &= -0.90617985 \\
w_2 &= 0.47862867 & x_2 &= -0.53846931 \\
w_3 &= 0.56888889 & x_3 &= +0.00000000 \\
w_4 &= 0.47862867 & x_4 &= +0.53846931 \\
w_5 &= 0.23692689 & x_5 &= +0.90617985
\end{align*}
\]  

(13.92)

Further, returning to the original variable, we note that

\[
\int_{a}^{b} g(t) \, dt = \sum_{i=1}^{m} \frac{\Delta t_i}{2} \sum_{j=1}^{5} w_j g \left( t_i^{\text{mid}} + \frac{\Delta t_i}{2} x_j \right)
\]

(13.93)

As an aside, note that the points \( x_1 \) and \( x_2 \) at which we have evaluated the function for two-point Gaussian quadrature are the two roots of the second Legendre polynomial, \( L_2(x) = \frac{1}{2}(3x^2 - 1) \) and the weight to be applied to \( f(x_i) \) is given by \((2/(1-x^2))(dL_2(x_i)/dx)^2\). More generally, for an \( N \)-point Gaussian integration, we would discover that\(^7\)

\[
L_N(x_i) = 0 \quad \text{and} \quad w_i = \frac{2}{(1-x_i^2)(dL_N(x_i)/dx)^2}
\]

(13.94)

\(^6\)The advantage isn’t that great, however, because, for Simpson’s rule, the upper evaluation for one strip could also be used as the lower evaluation for the next strip. No such feature applies to the two-point—or to any—Gaussian integration formula. The advantage of the Gaussian approach increases, however, as \( N \) in Eq. (13.80) increases.

and that
\[ \left| \int_{-1}^{1} f(x) \, dx - \sum_{k=1}^{N} w_k f(x_k) \right| = \frac{2^{2N+1}(N!)^4}{(2N+1)(2N)!^3} \left| \frac{d^{2N} f}{dx^{2N}} \right|_{x=\xi} \]

where \(-1 < \xi < 1\). This result shows that the \(N\)-point formula of this type will be 100\% accurate for polynomials of degree \(2N-1\) or lower—a property which we have already inferred informally. Because of the role played by the Legendre polynomials in these formulae, they are sometimes referred to as \textit{Gauss-Legendre formulae}.\(^8\)

### 13.14 Evaluating Integrals Numerically with C

C programs to evaluate integrals numerically can be constructed in several ways. In this section, we describe how we can implement one or another algorithm directly in a program that we write from scratch—a task that is tedious and difficult except for the simplest of algorithms. In addition, we describe how we can make use of available standard subroutines—we here focus on those in the \textit{Numerical Recipes} library—and devote our efforts solely to the easier task of writing a suitable \textit{driving program} to invoke the features of whatever integration subroutine we choose.

#### 13.14.1 Writing Programs from Scratch

\textit{Note:} All C programs (*.c) and all C-generated data files (*.c.dat) in this subsection can be copied from the directory \$HEAD/cc. \$HEAD for your site is defined in the \textit{Local Guide}.

More to illustrate the essence of an algorithm than to develop a truly useful integration program, we begin by presenting a quick C program to implement the trapezoidal rule as expressed in Eq. (13.55). The program contains three major sections. The first section loads standard library routines, declares variables, and obtains controlling input (integration limits and desired number of segments) and is expressed with the coding

```c
#include <stdio.h> /* Load standard i/o routines */
#include <math.h> /* Load standard math routines */
float a, b, value; /* For limits, sum */
float dx; /* For step size */
int n, i; /* For number of segments, loop index */
printf( "Lower limit : " ); scanf("%f", &a );
printf( "Upper limit : " ); scanf("%f", &b );
printf( "Number of segments: " ); scanf("%d", &n );
```

The second section evaluates the integral with the coding

```c
dx = (b-a)/n; /* Set size of segment */
value = 0.5*func(a); /* Compute first term */
for ( i=1; i<=n-1; i++ ) /* Add middle terms */
    value = value + func( a + i*dx );
```

\(^8\)A more general integral that can be approached with the techniques of this subsection has the form
\[
\int_{a}^{b} f(x) w(x) \, dx
\]
where \(w(x)\) is a weighting function. In the case we dealt with, \(w(x) = 1\) and, in our rescaling, the interval became the interval from \(-1\) to \(1\). That the Legendre polynomials \(L_{\ell}(x)\) are orthogonal on the interval \(-1 < x < 1\) with weight \(w(x) = 1\) is part of the reason that the roots of these polynomials ultimately emerged as important. For other weight functions and other intervals, a different set of polynomials would have played the role of the Legendre polynomials. Thus, there are several different types of Gaussian quadrature, each specific to a particular weight and basic interval.
Finally, the third section displays the resulting evaluation with the coding

```c
printf("Integral = %10.6f\n", value);
```

Partly because we would otherwise have to type the integrand three times and partly because we wish to make the program relatively flexible, we have defined the integrand as a user-defined function, whose definition is conveyed by the coding

```c
float func( float x ) /* Define integrand as function */
{
    return 2.0*exp(-pow(x,2))/sqrt(3.14159265);
}
```

which must be placed before the main program. A full listing of this program, named `trapezoidal.c`, is presented in Section 13.B. Simply changing the definition of the function `FUNC` will create a program that will integrate a different function.

For the sake of an explicit example, we have defined `func` in `trapezoidal.c` to return the integrand for the error function as defined in Eq. (13.26). To compile and run this program to evaluate erf(1.0) for various N, we execute the statements

```c
cc -o trapezoidal.xc trapezoidal.c -lm
./trapezoidal.xc
```

```
Lower limit : 0.0
Upper limit : 1.0
Number of segments: 1
Integral = 0.771743
```

The results for various divisions of the integration interval are shown in Table 13.1. Note in particular that, as N increases from 1, the results change noticeably (reflecting decreasing truncation error) until N reaches 256 or 512. From N = 256 to N = 65536, however, the results are quite stable. For N > 65536, the results once again change (this time reflecting increasing roundoff error). This table provides evidence that there indeed is a fairly wide range in which N is simultaneously large (Δx is small enough) to give accurate results but not so large (Δx not so small) as to trigger problems from computer roundoff. We conclude that, to six digits, the value of this integral is 0.842701.

This example, of course, illustrates only how to write a C program to address integrals that fall into our first category—integrals that are simply a number. If the integral of interest happens to depend on a parameter, our approach needs refinement. When the parameter occurs in the limits of integration, e.g., as the upper limit, we simply invoke the integrating routine—here all the statements in the second section identified at the beginning of this subsection—in a loop, specifying a different upper limit with each pass through the loop. When the parameter occurs embedded in the integrand, we still need a loop. In addition, however, to communicate the parameter from the main program to the function, we must either (1) add that parameter as an argument to the function or, alternatively, (2) arrange for the main program to store the parameter in a global variable (by declaring the variable to be used for the parameter `ahead of and outside` all other program blocks.

---

9To be specific, both here and throughout this section, we illustrate these steps with the statements that would be used in UNIX to compile, link, and run the program. Other operating systems probably accomplish the same end with different statements. In particular, preceding the program name with `./` is necessary in UNIX but may not be necessary with other operating systems. Details will be found in the Local Guide.

10Don't overlook the final item (`-lm`) in the first statement. It is necessary to tell the compiler to link in the C mathematical libraries.
Table 13.1: Values of \( \text{erf}(1.0) \) obtained with the trapezoidal rule and the C program \texttt{trapezoidal.c}.

<table>
<thead>
<tr>
<th>( n )</th>
<th>( T_n )</th>
<th>( n )</th>
<th>( T_n )</th>
<th>( n )</th>
<th>( T_n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.771743</td>
<td>256</td>
<td>0.842700</td>
<td>65536</td>
<td>0.842701</td>
</tr>
<tr>
<td>2</td>
<td>0.825263</td>
<td>512</td>
<td>0.842700</td>
<td>131072</td>
<td>0.842699</td>
</tr>
<tr>
<td>4</td>
<td>0.838368</td>
<td>1024</td>
<td>0.842701</td>
<td>262144</td>
<td>0.842694</td>
</tr>
<tr>
<td>8</td>
<td>0.841619</td>
<td>2048</td>
<td>0.842700</td>
<td>524288</td>
<td>0.842636</td>
</tr>
<tr>
<td>16</td>
<td>0.842431</td>
<td>4096</td>
<td>0.842701</td>
<td>1048576</td>
<td>0.842636</td>
</tr>
<tr>
<td>32</td>
<td>0.842633</td>
<td>8192</td>
<td>0.842701</td>
<td>2097152</td>
<td>0.839947</td>
</tr>
<tr>
<td>64</td>
<td>0.842684</td>
<td>16384</td>
<td>0.842701</td>
<td>4194304</td>
<td>0.847357</td>
</tr>
<tr>
<td>128</td>
<td>0.842696</td>
<td>32768</td>
<td>0.842701</td>
<td>8388608</td>
<td>0.847357</td>
</tr>
</tbody>
</table>

and then using the declared variable name both in the function and in the main program when values are assigned to the parameter).\(^{11}\)

### 13.14.2 Using Numerical Recipes

**Note:** CPSUP-generated C programs (*.c) and C-generated data files (*.c.dat) can be downloaded from the directory \$\text{HEAD}/cc. If your site has the appropriate license, files in this subsection from the Numerical Recipes library can be downloaded from a subdirectory of the directory \$\text{NRHEAD} (see Fig. 10.1). \$\text{NRHEAD} and \$\text{HEAD} for your site are defined in the Local Guide.

Numerical Recipes provides many integration procedures cast in the form of functions, including \texttt{trapzd.c}, \texttt{qtrap.c}, \texttt{qsimp.c}, and \texttt{qromb.c}. Since it can be used by itself and it is also used in the interior of the other routines, \texttt{trapzd.c} is the workhouse of the entire suite. It is called by the statement

\[
s = \text{trapzd}(\text{func}, a, b, i) ;
\]

When invoked, \texttt{trapzd.c} evaluates the integral of the function defined by \texttt{func} over the limits from \( a \) to \( b \), dividing that interval into \( 2^{i-1} \) segments\(^{12}\) and returning the result in the variable \( s \). Note, however, that this procedure must be called in a sequence of calls, starting with \( i = 1 \), then with \( i = 2, i = 3 \), etc. The variable \( s \) is initialized in the first call and then incremented appropriately with each subsequent call for the next larger value of \( i \). At base, \texttt{trapzd.c} evaluates \( T_1 \) and then—recursively—\( T_2, T_4, T_8, \ldots \), using the efficiency described in the first paragraph of Section 13.5.2.

A C driving program for using \texttt{trapzd.c} must be written by the user and will have the general form listed in Table 13.2.\(^{13}\) This program defines the function to be integrated, establishes the desired limits, invokes \texttt{trapzd} in a properly constructed loop, and displays each value as it is obtained.

For the sake of an explicit example, we have chosen to evaluate \( \text{erf}(1.0) = \frac{2}{\sqrt{\pi}} \int_0^1 e^{-x^2} dx \). The steps in evaluating the desired integral with this program then are

1. Create the driving program, either by copying the demonstration program \texttt{xtrapzd.c} from the directory\(^{14}\) \$\text{NRHEAD}/recipes_c-ansi/demo/src to the default directory, giving it the name

\[^{11}\text{Details on the use of global variables in C are discussed in Section 9.7.4; further illustration will be found in Section 11.15.}\]

\[^{12}\text{The parameter \( i \) cannot, however, be made too large, since roundoff errors become more significant as the number of segments becomes excessive. As a rule of thumb, \( i \) should probably be kept less than about 15.}\]

\[^{13}\text{A somewhat more elaborate demonstration program named \texttt{xtrapzd} can be found in the directory \$\text{NRHEAD}/recipes_c-ansi/recipes or—possibly—in the directory \$\text{NRHEAD}/recipes_c-kr/recipes. (See the Local Guide.)}\]

\[^{14}\text{At some sites, the KR variant of C may be installed rather than the ANSI variant. In that case, the directory from which all files should be copied will be \$\text{NRHEAD}/recipes_c-kr/...}\]
13.14. EVALUATING INTEGRALS NUMERICALLY WITH C

Table 13.2: The C program errtrapzd.c.

```c
/* PROGRAM errtrapzd */
#include <stdio.h>  /* Load standard i/o routines */
#include <math.h>   /* Load standard math functions */
#include "nr.h"     /* Load necessary recipes headers */

float func(float x)  /* Define integrand as function */
{
    return 2.0*exp(-pow(x,2))/sqrt(3.14159265);
}

main()
{
    float a, b, s;  /* For limits, sum */
    int numit, i;   /* For number of iterations */
    numit = 14;     /* Set number of iterations */
    a = 0.0; b = 1.0; /* Set limits */

    /* Head table, then invoke trapzd iteratively */
    /* and display result at each step */

    printf("%6s %24s
","n","approx. integral");
    for (i=1;i<=numit;i++){
        s=trapzd( func, a, b, i );
        printf("%6d %20.6f\n",i,s);
    }
}
```

errtrapzd.c, and editing it appropriately or by creating the file errtrapzd.c from scratch with an available text editor.

2. Copy the recipe trapzd.c from the directory $NRHEAD/recipes_c-ansi/recipes and the header file nr.h from the directory $NRHEAD/recipes_c-ansi/include to the default directory.

3. Compile and link the executable module with the statement

   ```
   cc -o errtrapzd.xc errtrapzd.c trapzd.c -lm
   ```

4. Run the program with the statement ./errtrapzd.xc.

The output from the program will be as shown in Table 13.3. Because the last several of these values do not change, we conclude that, to six digits, the value of this integral is 0.842701. This output also shows the way truncation error diminishes as the step size is decreased.

The Numerical Recipes function qtrap.c uses trapzd.c in a loop that evaluates the integral repeatedly, assessing convergence at each step, and stops either when two consecutive values differ by less than $10^{-3}\%$ of the first of the two [as defined by the editable parameter EPS (default $10^{-5}$) in the function qtrap.c] or when more than 20 refinements of the step size [as defined by the editable parameter JMAX (default 20)] have been effected without convergence.\textsuperscript{15} It is invoked with

\textsuperscript{15}In some cases, e.g., when the value of the integral being addressed is actually zero, it may be prudent to edit qtrap.c to replace the convergence criterion based on fractional change with one based on absolute change in consecutive iterates.
Table 13.3: Output from errtrapzd.c.

<table>
<thead>
<tr>
<th>n</th>
<th>approx. integral</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.771743</td>
</tr>
<tr>
<td>2</td>
<td>0.825263</td>
</tr>
<tr>
<td>3</td>
<td>0.838368</td>
</tr>
<tr>
<td>4</td>
<td>0.841619</td>
</tr>
<tr>
<td>5</td>
<td>0.842431</td>
</tr>
<tr>
<td>6</td>
<td>0.842633</td>
</tr>
<tr>
<td>7</td>
<td>0.842684</td>
</tr>
<tr>
<td>8</td>
<td>0.842697</td>
</tr>
<tr>
<td>9</td>
<td>0.842700</td>
</tr>
<tr>
<td>10</td>
<td>0.842701</td>
</tr>
<tr>
<td>11</td>
<td>0.842701</td>
</tr>
<tr>
<td>12</td>
<td>0.842701</td>
</tr>
<tr>
<td>13</td>
<td>0.842701</td>
</tr>
<tr>
<td>14</td>
<td>0.842701</td>
</tr>
</tbody>
</table>

Table 13.4: The C program errqtrap.c.

```c
/* PROGRAM errqtrap */

#include <stdio.h>  // Load standard i/o routines
#include <math.h>   // Load standard math routines
#include "nr.h"     // Load necessary recipes headers

float func(float x) /* Define integrand as function */
{
    return 2.0*exp(-pow(x,2))/sqrt(3.14159265);
}

main()
{
    float a, b, s;  // For limits, sum
    a = 0.0; b = 1.0;  // Set limits
    s = qtrap( func, a, b );  // Evaluate integral
    printf("Result using QTRAP = %10.6f\n", s);  // Display result
}
```

A statement of the form

```c
s = qtrap( func, a, b );
```

where `func` defines the function to be integrated, `a` and `b` are the lower and upper limits, and `s` is the variable in which the routine returns the final value. Thus, a driving program to invoke `qtrap.c` is simpler than one using `trapzd.c` directly and might have the form shown in Table 13.4. The steps in evaluating the desired integral with this program are

1. Create the driving program, either by copying the demonstration program `xqtrap.c` from the
directory giving it the name errqtrap.c and editing it appropriately, or by creating the file errqtrap.c from scratch with an available text editor.

2. Copy any necessary recipes and include files to the default directory. In the present case, the needed recipe trapzd.c and header file nr.h presumably remain from the first example. Each of the recipes qtrap.c and—discovered in a first (failed) attempt at compilation—nrutil.c must be copied from the directory $NRHEAD/recipes.c-ansi/recipes.

3. Compile and link the executable module with the statement

```
cc -o errqtrap.xc errqtrap.c trapzd.c qtrap.c nrutil.c -lm
```

4. Run the program with the statement ./errqtrap.xc.

The output from the program will be

```
Result using QTRAP = 0.842700
```

which is certainly in agreement with earlier results (Note that failure to converge will result in a message that includes “Too many steps ...” being printed on the screen. Here, the maximum number of steps is 20 by default, that value being defined by an editable parameter in the subroutine qtrap.c. After printing this message, the subroutine returns control to the operating system.)

The Numerical Recipes procedures qsimp.c and qromb.c are used in a way quite parallel to that of qtrap.c. The generation of programs to evaluate this test integral using each of these additional routines is left to the exercises. That these additional routines also yield the value 0.842701 is not at all a surprise.

Throughout this subsection, we have confined our explicit examples to integrals falling into our first category. If the integral of interest happens to depend on a parameter, our approach needs refinement. When the parameter occurs in the limits of integration, e.g., as the upper limit, we simply invoke the integrating routine in a loop, specifying a different upper limit with each pass through the loop. When the parameter occurs embedded in the integrand, we still need a loop. In addition, however, to communicate the parameter from the main program to the function, we must either (1) add that parameter as an argument to the function and edit the numerical recipes routine to accept that parameter as an argument as well or, alternatively, (2) arrange for the main program to store the parameter in a global variable (by declaring the variable to be used for the parameter ahead of and outside all other program blocks and then using the declared variable name both in the function and in the main program when values are assigned to the parameter).\(^{16}\)

### 13.15 Exercises

#### 13.15.1 ... using Symbolic Methods

13.1. A particle of mass \( m \) moves non-relativistically in one dimension under the action of a constant force \( f \). Starting with Eq. (13.3) and using symbolic integration, find the position \( x \), velocity \( v \), and momentum \( p \) of this particle as functions of time if \( x(0) = x_0 \) and \( v(0) = v_0 \).

13.2. A particle of mass \( m \) moves non-relativistically in one dimension under the action of a constant force \( f \). Starting with Eqs. (13.5) and (13.6) and using symbolic integration, find the position \( x \), velocity \( v \), and momentum \( p \) of this particle as functions of time if \( x(0) = x_0 \) and \( v(0) = v_0 \).

\(^{16}\)See footnote 11.
13.3. A particle of mass $m$ moves non-relativistically in one dimension $x$ under the action of a force given by $f(x) = -kx$, where $k$ is a (spring) constant. Starting with Eqs. (13.5) and (13.6) and using symbolic integration, find the position $x$, velocity $v$, and momentum $p$ of this particle as functions of time if $x(0) = x_0 > 0$ and $v(0) = 0$.

13.4. Suppose an object of mass $m$ moves non-relativistically in one dimension under the action of the force $f(t) = f_0 e^{-bt}$, where both $b$ and $f_0$ are positive. Let $x(0) = x_0$ and $v(0) = v_0$. Use symbolic integration to find $x(t)$ and $v(t)$ by evaluating the integrals in Eq. (13.3). Then, find and interpret both the limits of these two results as $t \to \infty$ and the Taylor expansion of these two results for small $t$.

13.5. The normalized Lorentz distribution function is given by

$$p(x) = \frac{1}{\pi} \frac{a/2}{x^2 + (a/2)^2}$$

Using symbolic integration, (a) verify that $\int_{-\infty}^{\infty} p(x) \, dx = 1$, (b) evaluate—as best you can—the average $\tau$ and variance $\sigma^2$, defined by

$$\tau = \lim_{b \to \infty} \int_{-b}^{b} x \, p(x) \, dx \quad \text{and} \quad \sigma^2 = \lim_{b \to \infty} \int_{-b}^{b} (x - \tau)^2 \, p(x) \, dx$$

for this distribution, and (c) find the probability that a single, randomly selected value will lie in the range $-a \leq x \leq a$. Finally, (d) show analytically that you should have expected the result of part (c) to be independent of $a$. Hint: Introduce the dimensionless variable $\lambda = x/a$.

13.6. Suppose some cataclysmic event stops the earth dead in its tracks and, responding to the sun’s gravitational attraction, the earth falls into the sun. Using symbolic integration, find the time required for the earth to fall over the middle half of its journey to the sun. Expressed in years, what is the value of this time for the earth-sun system? Hint: Since the gravitational potential is $-GmM/x$, conservation of energy yields

$$\frac{1}{2} m \left( \frac{dx}{dt} \right)^2 - G \frac{mM}{x} = -G \frac{mM}{x_0} \quad \Rightarrow \quad \frac{dx}{dt} = -\sqrt{2GM} \sqrt{\frac{1}{x} - \frac{1}{x_0}}$$

(The negative square root is taken because $x$, the distance to the sun, is known to be decreasing.) This expression then leads to the value

$$T_{\text{midhalf}} = \frac{1}{\sqrt{2GM}} \int_{x_0/4}^{3x_0/4} \left( \frac{1}{x} - \frac{1}{x_0} \right)^{-1/2} \, dx$$

Hint: The evaluation will be simpler if you begin by recasting the problem in dimensionless terms, expressing lengths in units of $x_0$ and times in units of $\sqrt{x_0^3/(2GM)}$. To interpret the significance of this unit of time, determine the period of a circular orbit of radius $x_0$, which will turn out to be $2\pi \sqrt{x_0^3/(2GM)}$. For the earth around the sun, this latter time is, of course, 1 year. Optional: Evaluate the time required for the first half of the journey, which involves a convergent but improper integral.

13.7. According to the quantum theory, the probability that the electron in the ground state of the hydrogen atom will be found between the center of the atom and some radius $r$ is given by

$$P(r) = \frac{4}{a^3} \int_0^r e^{-2r'/a} r' \, dr' = 4 \int_0^{r/a} e^{-2\rho} \rho^2 \, d\rho$$

where $a$ is the Bohr radius and $\rho = r'/a$. Evaluate this integral symbolically. Then plot and comment on a graph of $P(r)$ versus $r/a$.

13.8. Consider a source consisting of two uniformly charged disks, each of radius $a$ and each oriented with its center on the $z$ axis and its plane perpendicular to the $z$ axis. Let one disk have its center
at \((0,0,ca)\) and carry a positive charge density \(\sigma\) and the other have its center at \((0,0,-ca)\) and carry a negative charge density \(-\sigma\). Using a symbolic program, show that the on-axis electrostatic potential established by this source is given by

\[
V(z) = \frac{\sigma}{2\alpha} \left[ \sqrt{a^2 + (z-ca)^2} - |z-ca| - \sqrt{a^2 + (z+ca)^2} + |z+ca| \right]
\]

and then explore this potential as a function of \(z/a\) for various values of \(c\). Hint: First find the on-axis potential of a single disk lying in the \(xy\) plane, and then construct the desired potential by superposition.

13.9. In a spherically symmetric charge distribution, the charge density is a function of only the radial coordinate, \(\rho(r) = \rho(r)\). Suppose \(\rho(r) = 0\) for \(r > a\). Find the electrostatic potential at a point on the \(z\) axis, \(r = z\mathbf{k}\), for which \(z > a\) by setting up and evaluating the integral

\[
V(0,0,z) = \frac{1}{4\pi \varepsilon_0} \int \frac{\rho(r') \, dr'}{|r-r'|}
\]

symbolically to show that \(V(0,0,z) = Q/(4\pi \varepsilon_0 z)\). (\(Q\) is the total charge in the distribution.) This result demonstrates formally that the potential at a point outside a spherically symmetric charge distribution can be evaluated by regarding the charge to be concentrated at the center of the distribution. Hint: Write \(r' = x' + y' \mathbf{j} + z' \mathbf{k}\) but then use spherical coordinates to express \(x'\), \(y'\), and \(z'\).

13.10. Consider a surface in the \(xy\) plane having uniform mass density \(\sigma\) and having the shape of a cardioid given in polar coordinates by the function \(r(\phi) = a(1 - \cos \phi)\). Using symbolic integration, find (a) the center of mass of this object, (b) the moment of inertia tensor of this object about the \(x\), \(y\), and \(z\) axes, and (c) the radius of gyration about the \(z\) axis. Hints: The center of mass is defined in Section 13.1.2; the moment of inertia tensor is a \(3 \times 3\) tensor whose \(ij\) element is given by

\[
I_{ij} = \int [(x_1^2 + x_2^2 + x_3^2) \delta_{ij} - x_i x_j] \, dm
\]

where \(x_1\), \(x_2\), and \(x_3\) symbolize \(x\), \(y\), and \(z\), respectively; \(\delta_{ij}\) is the Kronecker delta, which has the value 1 when \(i = j\) and the value 0 otherwise; and the radius of gyration is defined in Section 13.1.3.

13.11. In quantum mechanics, the two integrals

\[
x_{mn} = \int_{-\infty}^{\infty} \psi_m^*(x) x \psi_n(x) \, dx \quad \text{and} \quad p_{mn} = \int_{-\infty}^{\infty} \psi_m^*(x) \frac{\hbar}{i} \frac{\partial}{\partial x} \psi_n(x) \, dx
\]

are important in a variety of contexts. For a particle in an infinitely deep potential well that extends over the region \(-a \leq x \leq a\),

\[
\psi_n(x) = \begin{cases} 
\frac{1}{\sqrt{a}} \cos \frac{n\pi x}{2a} & n = 1, 3, 5, \ldots \\
\frac{1}{\sqrt{a}} \sin \frac{n\pi x}{2a} & n = 2, 4, 6, \ldots
\end{cases}
\]

Using symbolic integration, show that, for these wave functions, \(x_{mn} = 0\) and \(p_{mn} = 0\) when \(m\) and \(n\) are both even or both odd and that

\[
x_{mn} = \frac{16a}{\pi^2} \, (-1)^{(m+n+1)/2} \frac{mn}{(m^2 - n^2)^2}
\]

\[
p_{mn} = \frac{2\hbar}{a} \, (-1)^{(m+n+1)/2} \frac{mn}{(m^2 - n^2)}
\]

otherwise. Note that, for purposes of translating the general integrals above to the circumstances of this exercise, the wave functions should both be regarded as zero outside of the interval \(-a \leq x \leq a\).
13.12. The sawtooth wave is defined by

\[ f(x) = \frac{x}{l} ; \quad -l \leq x \leq l \]

Use symbolic integration to find the Fourier coefficients \(a_n\) and \(b_n\) for this function and then generate graphs showing the function given by the truncated series

\[ f_{\text{trunc}}(x) = \frac{a_0}{2} + \sum_{n=1}^{N} \left( a_n \cos \frac{n\pi x}{l} + b_n \sin \frac{n\pi x}{l} \right) \]

for various values of \(N\), including 0, 1, 2, 3, and 10.

13.13. The Legendre polynomials \(L_n(x)\), \(n = 0, 1, 2, \ldots\), are orthogonal on the interval \(-1 \leq x \leq 1\) with weight 1. In particular,

\[ \int_{-1}^{1} L_m(x) L_n(x) \, dx = \frac{2}{2n+1} \delta_{mn} \]

Any function defined over the interval \(-1 \leq x \leq 1\) can then be expanded in the Legendre series

\[ f(x) = \sum_{n=0}^{\infty} c_n L_n(x) \]

(a) Show by hand that the coefficient \(c_n\) in this expansion is given by

\[ c_n = \frac{2n+1}{2} \int_{-1}^{1} f(x) L_n(x) \, dx \]

(b) Use symbolic integration to find \(c_n\) for \(n = 0, 1, 2, 3, 4, 5\) and 6 in the Legendre expansion for the function

\[ f(x) = \begin{cases} 
-1 & -1 < x < 0 \\
1 & 0 < x < 1 
\end{cases} \]

(c) Graph the functions defined by the partial sums \(\sum_{n=0}^{N} c_n L_n(x)\) for \(N = 0, 1, 2, 3, 4, 5,\) and 6.

Hint: Quite possibly the symbolic program you are using has the Legendre polynomials built in somehow, and you should study its manuals to find out how to invoke them. Just in case that isn’t true, the first nine Legendre polynomials are

\[
\begin{align*}
L_0(x) &= 1 \\
L_1(x) &= x \\
L_2(x) &= \frac{1}{2}(3x^2 - 1) \\
L_3(x) &= \frac{1}{4}(5x^3 - 3x) \\
L_4(x) &= \frac{1}{8}(35x^4 - 30x^2 + 3) \\
L_5(x) &= \frac{1}{8}(63x^5 - 70x^3 + 15x) \\
L_6(x) &= \frac{1}{4}(231x^6 - 315x^4 + 105x^2 - 5) \\
L_7(x) &= \frac{1}{16}(429x^7 - 693x^5 + 315x^3 - 35x) \\
L_8(x) &= \frac{1}{128}(6435x^8 - 12012x^6 + 6930x^4 - 1260x^2 + 35) \\
L_9(x) &= \frac{1}{256}(12155x^9 - 25740x^7 + 18018x^5 - 4620x^3 + 315x)
\end{align*}
\]

13.15.2 … using Numerical Methods

13.14. To deduce Simpson’s rule, we start by supposing three consecutive values \(f_1, f_2,\) and \(f_3\) of the integrand, where for simplicity in notation we take the points of evaluation to be \(x_1 = x_2 - \Delta x, x_2,\) and \(x_3 = x_2 + \Delta x.\) Using a symbol manipulating program to do the algebra and calculus, (a) find the coefficients \(A, B,\) and \(C\) needed to make the parabola \(Ax^2 + Bx + C\) pass through the three points \((x_i, f_i), i = 1, 2, 3,\) (b) integrate that parabola over the interval \(x_1 < x < x_3\) to find that

\[ \int_{x_1}^{x_3} f(x) \, dx \approx \int_{x_1}^{x_3} (Ax^2 + Bx + C) \, dx = \frac{\Delta x}{3} \left( f_1 + 4f_2 + f_3 \right) \]

(c) show that this result actually gives the correct value for \(f(x) = x^3\) and, finally, (d) deduce the (extended) Simpson’s rule of Eq. (13.56). Note: Because this exercise relates to numerical algorithms, it has been placed in with other exercises that are numerical. This exercise is symbolic, and you should use a symbol manipulating program for parts (a), (b), and (c); however, you should address part (d) by hand.
13.17. (a) Deduce the three-point Gaussian formula, for which—paralleling Eq. (13.81)—we set
\[ \int_{-1}^{1} f(x) dx = w_1 f(x_1) + w_2 f(x_2) + w_3 f(x_3) \]

We then choose the six quantities \( w_i \) and \( x_i \) so that the expression gives an exact result for \( f(x) = 1, x, x^2, x^3, x^4, \) and \( x^5 \). (b) Verify that the interpolation points \( x_i \) and weights \( w_i \) are given by Eq. (13.94) where \( L_4(x) = (5x^3 - 3x)/2 \).

13.18. The (normalized) wave functions for a quantum harmonic oscillator in its first and second excited states (\( n = 1, n = 2 \)) are
\[ \psi_1(x) = \sqrt{2} \left( \frac{m\omega}{\pi\hbar} \right)^{1/4} y e^{-y^2/2}; \quad \psi_2(x) = \frac{1}{\sqrt{2}} \left( \frac{m\omega}{\pi\hbar} \right)^{1/4} (2y^2 - 1) e^{-y^2/2} \]
where \( y = x/\sqrt{\hbar/\omega} \), the energies of these states are \( 3h/2 \), and \( 5h/2 \), respectively, and the symbols have the same meanings as in Section 13.1.8. Find the probability that a harmonic oscillator in each of these states will be found outside the classical turning point.

13.19. The Maxwell-Boltzmann speed distribution yields the integral
\[ f(v) = 4\pi \left( \frac{m}{2\pi kT} \right)^{3/2} \int_v^\infty e^{-mv^2/2kT} v^2 dv \]
for the fraction of the molecules having speed less than \( v \). Using numerical means, explore this integral as a function of \( v \). \( \text{Hint:} \) Re-express the integral using \( \sqrt{2kT/m} \) as the unit of velocity.

13.20. Planck's black body radiation law gives the expression
\[ I(\nu_2, \nu_1) = \frac{8\pi h}{c^3} \int_{\nu_1}^{\nu_2} \frac{\nu^3}{e^{h\nu/kT} - 1} d\nu \]
for the power radiated per unit area in the frequency range \( \nu_1 \leq \nu \leq \nu_2 \). Using numerical means, explore the power radiated in the visible spectrum \( 4 \times 10^{14} \) Hz \( \leq \nu \leq 7 \times 10^{14} \) Hz as a function of temperature. \( \text{Hint:} \) One way to approach this exercise would be to choose a reference frequency \( \nu_0 \) arbitrarily (say \( 10^{14} \) Hz) and recast the integral on the dimensionless variable \( s = \nu/\nu_0 \). Examination of \( I \) in units of \( 8\pi h\nu_0^3/c^3 \) as a function of \( T \) in units of \( h\nu_0/k \) would then be indicated.

13.21. As used in statistical data analysis, the Gaussian distribution for a variable \( t \) is usually expressed in terms of the standard deviation \( \sigma \), the distribution function being
\[ \frac{1}{\sqrt{2\pi} \sigma} e^{-t^2/(2\sigma^2)} \]
Thus, the probability of finding a value between \( a \) and \( b \) is given by
\[ P(a, b) = \frac{1}{\sqrt{2\pi} \sigma} \int_a^b e^{-t^2/(2\sigma^2)} dt \]
Show analytically that \( P(-\sigma, \sigma) = \text{erf}(x/(\sqrt{2}\sigma)) \), and then evaluate \( P(-\sigma, \sigma), P(-2\sigma, 2\sigma), \) and \( P(-3\sigma, 3\sigma) \) numerically. The values of these three quantities are 0.6827, 0.9545, and 0.9973, respectively—values that give rise to the designations of 68%, 95%, and 99% confidence intervals in statistical data analysis.

13.22. Suppose some cataclysmic event stops the earth dead in its tracks and, responding to the sun’s gravitational attraction, the earth falls into the sun. Using numerical integration, find the time required for the earth to fall over the middle half of its journey to the sun. Expressed in years, what numerically is the value of this time for the earth-sun system? \( \text{Hint:} \) Since the gravitational potential is \( -GmM/x \), conservation of energy yields
\[ \frac{1}{2} m \left( \frac{dx}{dt} \right)^2 - G\frac{mM}{x} = -G\frac{mM}{x_0} \quad \Rightarrow \quad \frac{dx}{dt} = -\sqrt{2GM} \sqrt{\frac{1}{x} - \frac{1}{x_0}} \]
CHAPTER 13. EVALUATING INTEGRALS

According to the quantum theory, the probability that the electron in the ground state of the hydrogen atom will be found between the center of the atom and some radius \( r \) is given by

\[
P(r) = \frac{4}{a^3} \int_0^r e^{-2r'/a} r'^2 d r' = 4 \int_0^{r/a} e^{-2\rho^2} \rho^2 d\rho
\]

where \( a \) is the Bohr radius and \( \rho = r'/a \). Using numerical integration, evaluate this integral as a function of its upper limit. Then plot and comment on a graph of \( P(r) \) versus \( r/a \).

Explore these integrals as functions of the modulus \( k \). As part of your exploration, obtain a graph of the period \( T \) of a simple pendulum as a function of the amplitude \( \alpha \) of that pendulum. Analytically, the period of that pendulum is given as a function of \( \alpha \) by \( T/T_0 = (2/\pi)K(\sin(\alpha/2)) \), where \( T_0 \) is the period of the pendulum at small amplitude.

The angular position \( \theta(t) \) of a simple pendulum swinging with amplitude \( \alpha \) is given by the integral

\[
\omega t = \int_0^\beta \frac{d\phi}{(1-k^2 \sin^2 \phi)^{1/2}}
\]

where, with \( l \) the length of the pendulum and \( g \) the acceleration of gravity, \( \omega = \sqrt{g/l} \), \( k = \sin(\alpha/2) \), and \( \beta = \sin^{-1}(\sin(\theta/2)/k) \). Remember that, because of the choice of signs (see Section 13.1.4), this integral is valid only during the portion of the swing from \( \theta = 0 \) to \( \theta = \alpha \). Obtain graphs of \( \theta \) versus \( \omega t \) over the first quarter of the pendulum’s swing for several different values of \( \alpha \).

The \( n \)-th order Bessel function can be defined by the integral

\[
J_n(x) = \frac{1}{\pi} \int_0^\pi \cos(n\theta - x \sin \theta) d\theta
\]

By evaluating this integral numerically as a function of \( x \) for different values of \( n \), obtain graphs of \( J_0(x) \), \( J_1(x) \), and \( J_2(x) \) over the range \( 0 \leq x \leq 10 \).
13.28. The Bessel function $J_1(x)$ can be defined by the integral

$$\frac{1}{x} J_1(x) = \frac{2}{\pi} \int_0^1 (1 - u^2)^{1/2} \cos(xu) \, du$$

Using this definition, obtain a graph of $J_1(x)$ versus $x$ over the range $0 \leq x \leq 10$.

13.29. A circular ring of radius $a$ resides in the $xy$ plane with its center at the origin and carries a charge $Q$ uniformly distributed about its perimeter. The electrostatic potential established by this ring at an observation point whose cylindrical coordinates are $(r, \phi, z)$ is

$$V(r, \phi, z) = \frac{Q}{4\pi \epsilon_0 a} \left( 1 - \frac{r}{a} \cos \phi' + \frac{z^2}{a^2} - \frac{2ax \cos \phi'}{a^2} \right)^{-1/2} d\phi'$$

Explore this integral as a function of $r/a$ for several values of $z/a$.

13.30. A circular current loop of radius $a$ lies in the $xy$-plane with its center at the origin and carries a current $I'$ counterclockwise as viewed from a point on the positive $z$ axis. The magnetic field at a point in the $xz$ plane is given by

$$\mathbf{B}(x, z) = \frac{\mu_0 I'/2\pi a}{2} \left[ \int_{x_a}^x \frac{z \cos \phi' \hat{i} + (a - x \cos \phi') \hat{k}}{x^2 + z^2 + a^2 - 2ax \cos \phi'} \, d\phi' \right]$$

Explore both components of this magnetic field numerically as functions of $x/a$ for various values of $z/a$, including $z/a = 0$ (which will require some creativity for dealing with the point $x/a = 1.0$, at which the integrand diverges at one point in the range of the integration variable).

13.31. In a dimensionless presentation, the intensity in the Fresnel diffraction pattern produced by a single slit when that slit is illuminated by a line source parallel to the slit is proportional to the quantity

$$I \propto \left| \int_{t_b}^{t_b + \delta t} e^{i \pi t^2/2} \, dt \right|^2$$

where $t$ is measured in a unit determined by the distance of the source from the screen containing the slit, the distance of the observation point from that same screen, and the wavelength of the illuminating radiation. In these units, $t_b$ locates the position of the lower edge of the slit (or, equivalently, the observation point in the diffraction pattern) and $\delta t$ measures the width of the slit. Obtain graphs of $I$ versus $t_b$—i.e., graphs of $I$ versus position on the viewing screen—for various values of $\delta t$. Hints: (1) As a start, let $t_b$ range over the interval $-4 \leq t_b \leq 4$ and examine values of $\delta t$ on the order of 1, but allow these initial explorations to suggest possibly more appropriate values. (2) Note that the real and imaginary parts of the integral appearing in this exercise are related to the integrals defining the Cornu spiral discussed in Section 13.1.6.

13.34. Patterning your program after the program trapezoidal.c, write and test a C program that uses Simpson’s rule [Eq. (13.56)] to evaluate

$$I = \frac{2}{\sqrt{\pi}} \int_a^b e^{-x^2} \, dx$$

13.35. Write and test a C program that uses the five-point Gaussian formula to evaluate the integral

$$I = \frac{2}{\sqrt{\pi}} \int_a^b e^{-x^2} \, dx$$
13.15.3 ... using Numerical Recipes

Note: Numerical recipes can, of course, also be used for any of the exercises in Section 13.15.2.

13.43. Edit the program `errtrapzd.c` so that it requests the input of the limits and the number of iterations at execution time, prints only the final value obtained, and does not report each value along the way. Be sure to test your program.

13.44. Following the pattern illustrated in Section 13.14.2, create and test C programs `errqsimp.c` and `errqromb.c` using `qsimp.c` and `qromb.c`, respectively, to evaluate erf(1.0). Note that, in addition to `trapzd.c`, `qromb.c` also invokes `polint.c`, which will have to be available and included in the compile instruction before `errqromb.c` will compile successfully.

13.45. Edit the driving program `errqtrap.c` (or—better—the driving program `errqsimp.c` created in the previous exercise) so that it requests the limits of integration as input from the terminal at execution time, prints the limits and the value of the integral, and then returns to the beginning to request a new set of limits. Include a smooth way (other than simply CONTROL-C) to terminate execution of the program. Be sure to test your program.

13.46. Edit the driving program `errqtrap.c` (or—better—the driving program `errqsimp.c` created in an earlier exercise) so that it evaluates the integral

\[ F(x) = \int_{0}^{x} g(s) \, ds \]

as a function of the upper limit \( x \), printing a table of values of \( x \) and \( F(x) \) for values of \( x \) ranging in steps \( \Delta x \) from \( x = a \) to \( x = b \), where \( g(s) \) is defined by a function subprogram and \( \Delta x \), \( a \), and \( b \) are to be entered at execution time. Use the error function \( \text{erf}(x) \) as defined in Eq. (13.26) as a test integrand for your program.

13.47. Use `trapzd.c`, `qtrap.c`, `qsimp.c`, and `qromb.c` in a succession of programs to evaluate the quantity

\[ f(v) = \int_{0}^{v} v^2 e^{-v^2/2} \, dv \]

(a) for \( v = 1.0 \) and (b) as a function of \( v \) over the range \( 0.0 \leq v \leq 3.0 \). Cast your response to part (b) so that it writes the values of \( v \) and \( f(v) \) into a (text) file that could then be imported into another program for graphical display. Statements for opening, writing to, and closing a file in C are introduced in Chapter 9.

13.48. Use Numerical Recipes in C to address Exercise 13.19, which involves evaluating an integral as a function of its upper limit.

13.49. Use Numerical Recipes in C to address Exercise 13.27, which involves evaluating an integral as a function of a parameter in the integrand. You will, of course, have to refresh your understanding of global variables in C.
13.B Listing of trapezoidal.c

/* PROGRAM trapezoidal.c

This C program evaluates the integral of a user-specified function between user-specified limits for a user-specified number of divisions of the interval of integration. Integration is achieved with the trapezoidal rule.
*/

#include <stdio.h> /* Load standard i/o routines */
#include <math.h> /* Load standard math routines */

float func( x ) /* Define integrand as function */
float x;
{
    return 2.0*exp(-pow(x,2))/sqrt(3.14159265);
}

main()
{
    float a, b, value; /* For limits, sum */
    float dx; /* For step size */
    int n, i; /* For number of segments, loop index */

    /***** Read limits, number of segments *****/
    printf( "\nLower limit    : " ); scanf("%f", &a);
    printf( "Upper limit : " ); scanf("%f", &b);
    printf( "Number of segments: " ); scanf("%d", &n);

    /***** Evaluate integral *****/
    dx = (b-a)/n; /* Set size of segment */
    value = 0.5*func(a); /* Compute first term */
    for ( i=1; i<=n-1; i++ ) /* Add middle terms */
        value = value + func( a + i*dx );
    value = value + 0.5*func(b); /* Add last term */
    value = value * dx; /* Compute integral */

    /***** Display result *****/
    printf("Integral = %10.6f\n", value);
}
Chapter 14

Finding Roots

In this chapter, we seek the roots of a known function \( f(x) \), i.e., we seek values of \( x \) satisfying the equation

\[
 f(x) = 0 \quad (14.1)
\]

If \( f(x) \) is simple, we may be able to find a closed form, analytic solution. More often, however, \( f(x) \) is sufficiently complicated that approximate, numerical methods are needed. Further, many functions will have several roots, some of which may be physically meaningless. Thus, we must learn not only how to find the roots but also how to sort the physically meaningful roots from a possibly larger number of mathematically acceptable ones, the “extras” of which are said to be spurious. We begin this chapter by identifying several physical situations, the full addressing of which requires finding one or more roots of some function. Then we illustrate how to use symbolic algebra systems to approach those that can be addressed analytically, describe a few of many available numerical algorithms, and describe ways to find roots using a variety of numerical approaches and computational tools. Briefly at the end, we comment about the more complicated issue of finding roots of sets of simultaneous linear and non-linear equations. Until that point, our discussion will focus on functions of a single variable.

Whatever the function and whatever the approach, the first step in seeking roots should always be to learn as much as possible about the nature of the function and its roots. Further, since numerical methods in particular—most of them iterative—require a starting guess or guesses and will converge more or less rapidly and reliably depending on the quality of those guesses, a priori knowledge of the approximate location of roots is essential. Thus, we should always start by drawing a graph of \( f(x) \) in sufficient detail to reveal the approximate location of the roots of interest. Since the focus of this chapter is not graphing (and graphing has been fully addressed in earlier chapters), we shall carry out this step once in Section 14.1 as we present several sample problems rather than carrying it out repeatedly in later sections.\(^1\)

14.1 Sample Problems

In this section, we identify several physical contexts in which the essential computational problem is to find the roots of some function, and we obtain graphs of the appropriate functions for later reference.

\(^1\)The graphs could, of course, be produced in any number of ways. Except for Fig. 14.2 (which was produced with \textit{tgif}), the graphs in Section 14.1 have all been produced with IDL. Those in each later section have been produced by whatever software is the subject of that section.
14.1.1 Classical Turning Points

Let \( V(x) \) be the potential energy under which an object of mass \( m \) is moving in one dimension. Turning points in the motion occur at values of \( x \) where the total energy \( E \) is entirely potential energy (kinetic energy is zero), i.e., when

\[
V(x) = E \quad \text{or} \quad V(x) - E = 0 \tag{14.2}
\]

Finding physical turning points thus involves finding the mathematical roots of the function \( f(x) = V(x) - E \), i.e., finding solutions to Eq. (14.2). If, for example, the potential energy of interest is given by the cubic polynomial

\[
V(x) = \frac{x^3}{10000} + \frac{x^2}{200} - \frac{x}{500} - \frac{1}{2} \tag{14.3}
\]

the turning points for the motion of a particle with total energy \( E = 0 \) moving in this potential energy would satisfy \( V(x) = 0 \). The first step in finding those turning points would therefore be to produce the graph of Fig. 14.1—a task that may require a bit of trial and error before a suitable range for the independent variable has been found. From this graph, we conclude that \( V(x) \) has three real roots, one in the vicinity of \( x = -50 \), a second in the vicinity of \( x = -10 \), and a third in the vicinity of \( x = 10 \). With more refined graphs drawn in the vicinity of each of these roots, we could conclude that the three roots are more tightly bound by the limits

\[
-50.00 < x_1 < -47.50 \quad ; \quad -12.50 < x_2 < -10.00 \quad ; \quad 7.50 < x_3 < 10.00 \tag{14.4}
\]

14.1.2 A Max-Min Problem: Equilibrium Points

From a different perspective, a particle moving in one dimension under the action of the force \( F = -dV/dx \) associated with the potential energy \( V(x) \) will be in equilibrium at those points \( x \) at which the force is zero, i.e., where

\[
F = 0 \implies \frac{dV}{dx} = 0 \tag{14.5}
\]

or where the graph of \( V(x) \) has a horizontal tangent. Further, evaluated at a point of equilibrium, \( d^2V/dx^2 > 0 \) implies that the equilibrium is stable while \( d^2V/dx^2 < 0 \) implies that the equilibrium is unstable. Finding points of physical equilibrium therefore involves finding mathematical roots of the function \( F(x) = -dV/dx \), and assessing the stability of those equilibria entails examining the sign of \( d^2V/dx^2 \).

More specifically, for the potential energy given in Eq. (14.3) and graphed in Fig. 14.1, we would find the points of equilibrium by solving the equation

\[
\frac{dV}{dx} = \frac{3x^2}{10000} + \frac{x}{100} - \frac{1}{500} = 0 \tag{14.6}
\]

By looking at local extrema in the graph of \( V(x) \), we infer that this potential energy exhibits two points of equilibrium, one located in the vicinity of \( x = -35 \) and the other in the vicinity of \( x = 0 \). A more refined graph leads to the conclusion that these two roots are bounded by

\[
-35.0 < x_4 < -30.0 \quad \text{and} \quad -2.5 < x_5 < 2.5 \tag{14.7}
\]

\(^2\)This function is, of course, not particularly realistic as a potential energy. We can, however, provide a physical context for at least a portion of \( V(x) \). If we confine our attention to the region around the one minimum it possesses, we can interpret the function as the potential energy of an anharmonic oscillator, for which—if that minimum occurs at \( x = 0 \)—we might write \( V(x) = kx^2/2 + ax^3 \) while imposing the constraint that \( a \), the coefficient of the cubic perturbation from the potential energy of a simple harmonic oscillator, be small. The expression in Eq. (14.2) simply places the minimum at a different value of \( x \) and adds a constant to the potential energy. In what follows, we will explore this function over a wider range of values of \( x \) than is physically meaningful. The pedagogic advantage of Eq. (14.2) is that it combines many of the important features of potential energy functions with, as we shall see, tractability by a variety of different approaches.
14.1.3 Natural Frequencies of Oscillating Systems

Suppose we seek the natural frequencies of oscillation for the system shown in Fig. 14.2, which consists of two objects, each having mass \( m \). Let these objects move in one dimension on a horizontal, frictionless surface, let them be connected to one another with a spring having constant \( k' \), and let each be connected to the nearer wall with a spring having constant \( k \). Further, let the position of each be measured from its equilibrium position. Then, Newton’s second law combined with Hooke’s law leads to the equations of motion\(^3\)

\[
m \frac{d^2x_1}{dt^2} = -kx_1 + k'(x_2 - x_1) \quad \text{and} \quad m \frac{d^2x_2}{dt^2} = -kx_2 - k'(x_2 - x_1) \tag{14.8}
\]

To cast these equations in dimensionless form, we choose a unit of length \( a \), set \( x_i/a = \overline{x}_i \), introduce \( \omega_0 = \sqrt{k/m} \), set \( \kappa = k'/k \), and introduce \( \overline{t} = \omega_0 t \) to find that

\[
\frac{d^2\overline{x}_1}{d\overline{t}^2} = -(1 + \kappa)\overline{x}_1 + \kappa \overline{x}_2 \quad \text{and} \quad \frac{d^2\overline{x}_2}{d\overline{t}^2} = \kappa \overline{x}_1 - (1 + \kappa)\overline{x}_2 \tag{14.9}
\]

Next, seeking sinusoidal (or simple harmonic) solutions, we suppose that

\[
\overline{x}_1(\overline{t}) = \overline{x}_{10} \cos \omega \overline{t} \quad \text{and} \quad \overline{x}_2(\overline{t}) = \overline{x}_{20} \cos \omega \overline{t} \tag{14.10}
\]

\(^3\)These equations were also discussed in Section 11.1.6.
where the (yet to be determined) frequency $\omega$ is measured in units of $\omega_0$. Substituting these suppositions into Eq. (14.9), we conclude that the (presently unknown) amplitudes must satisfy

$$
\left( \begin{array}{cc}
1 + \kappa - \omega^2 & -\kappa \\
-\kappa & 1 + \kappa - \omega^2
\end{array} \right) \left( \begin{array}{c}
\pi_{10} \\
\pi_{20}
\end{array} \right) = 0 \quad (14.11)
$$

The solution of this equation for the unknowns $\pi_{10}$ and $\pi_{20}$ will be trivial (i.e., both zero—a particularly uninteresting motion) unless the determinant of the matrix of coefficients happens itself to be zero, i.e., unless the characteristic equation

$$
D(\omega) = \begin{vmatrix}
1 + \kappa - \omega^2 & -\kappa \\
-\kappa & 1 + \kappa - \omega^2
\end{vmatrix} = (1 + \kappa - \omega^2)^2 - \kappa^2 = \omega^4 - 2\omega^2(1 + \kappa) + 1 + 2\kappa = 0 \quad (14.12)
$$

is satisfied. We arrive at a fourth-order polynomial $D(\omega)$ in $\omega$, the roots of which will give the natural frequencies of oscillation for the simple system under consideration. Since only even powers of $\omega$ appear, however, we can for the sake of a simpler solution regard the polynomial as quadratic in $\omega^2$.

Suppose we seek the dependence of the roots of this polynomial on $\kappa = k'/k$, i.e., on the strength of the middle spring compared to that of the two outer ones. Graphs of $D(\omega)$ versus $\omega$ for four different values of $\kappa$ are shown in Fig. 14.3. The lower of the two (positive) roots of $D(\omega)$ appears to be $\omega = 1$ regardless of the value of $\kappa$, while the upper of the two roots increases steadily as $\kappa$ increases.

### 14.1.4 Range of Projectile in Viscous Medium

For a fourth example, suppose we seek the range of a projectile of mass $m$ fired with the initial speed $v_0$ at angle $\theta$ up from the horizontal in a viscous medium having damping constant $b$. Supposing the
projectile to move in the \(xy\) plane, with \(x\) horizontal and \(y\) vertical, we begin by invoking Newton’s second law\(^4\) to write the equations of motion

\[
\frac{d^2x}{dt^2} = -b\frac{dx}{dt} \quad \text{and} \quad \frac{d^2y}{dt^2} = -mg - b\frac{dy}{dt}
\] (14.13)

Here, \(g\) is the acceleration of gravity (which we take to be a positive number). These equations are to be solved subject to the initial values

\[
x(0) = 0 \quad , \quad v_x(0) = v_0 \cos \theta \quad , \quad y(0) = 0 \quad \text{and} \quad v_y(0) = v_0 \sin \theta
\] (14.14)

The solution to this problem—see Chapter 11 on ordinary differential equations—is

\[
x(t) = \frac{mv_0 \cos \theta}{b} \left(1 - e^{-bt/m}\right)
\] (14.15)

\[
y(t) = -\frac{mgt}{b} + \frac{m}{b} \left(v_0 \sin \theta + \frac{mg}{b}\right) \left(1 - e^{-bt/m}\right)
\] (14.16)

Finally, to find the range \(R\), we seek the value of \(x\) at that non-zero value of \(t\) for which \(y(t) = 0\). Thus, we need to find the non-zero solution, say \(t_1\), of the equation

\[
-\frac{mgt}{b} + \frac{m}{b} \left(v_0 \sin \theta + \frac{mg}{b}\right) \left(1 - e^{-bt/m}\right) = 0
\] (14.17)

and then evaluate \(x(t_1)\). Equivalently, if we introduce the dimensionless time \(\tau = bt/m\) and the dimensionless parameter \(\alpha = bv_0/(mg)\), the equation whose root we seek becomes

\[
f(\tau) = \tau - \left(1 + \alpha \sin \theta\right) \left(1 - e^{-\tau}\right) = 0
\] (14.18)

Once we have found the desired root, say \(\tau_1\), we then determine the range \(R\) by substituting \(\tau_1\) into Eq. (14.15), finding that

\[
\frac{R(\theta, \alpha)}{v_0/g} = \frac{\cos \theta \left(1 - e^{-\tau_1}\right)}{\alpha} = \frac{\cos \theta}{\alpha(1 + \alpha \sin \theta)} \tau_1
\] (14.19)

The mathematical task confronting us involves finding the non-zero root of the function \(f(\tau)\) defined in Eq. (14.18). More specifically, finding the angle \(\theta\) at which the projectile should be fired to achieve maximum range would entail

- Choosing a value of \(\alpha = bv_0/(mg)\).
- Finding the non-zero root of \(f(\tau)\) for several values of \(\theta\) ranging from 0 to \(\pi/2\).
- Calculating and plotting values of \(R(\theta, \alpha)\).
- Finding the value of \(\theta\) corresponding to the peak in a graph of \(R(\theta, \alpha)\) versus \(\theta\).

Note that \(\alpha\) increases as \(b\) (the damping) increases and as \(v_0\) increases but decreases as \(m\) increases.

To gain some insight into the nature of the desired roots, we begin by plotting the family of graphs of \(f(\tau)\) versus \(\tau\) for various values of \(\theta\) with fixed \(\alpha\). After some exploration using techniques introduced in earlier chapters, we arrive at the several graphs shown in Fig. 14.4. Every graph exhibits a root at \(\tau = 0\), corresponding to the moment of launch. Though it is hard to judge, each graph for \(\theta = 0^\circ\) also exhibits a second root at \(\tau = 0\), since a projectile launched at \(\theta = 0^\circ\) returns to its initial altitude immediately. As the angle of launch is increased in each case, however, the second root moves to larger and larger values of \(\tau\). Our task in subsequent sections will be to find numerical values for that second root for various values of \(\alpha\) and \(\theta\), determine the range for each, and find the particular angle at which the range is greatest for a given \(\alpha\).

\(^4\)Compare Eqs. (11.3) and (11.4).
Figure 14.4: Graphs of \( f(\tau) \) versus \( \tau \). The individual frames in this display correspond to different values of \( \alpha \); the graphs in each frame correspond to different values of \( \theta \), ranging in 15° increments from 0° for the highest graph to 90° for the lowest graph in each frame. The scalings have been chosen to reveal the nature of the roots most clearly and are different in each frame.

As a quick aside, we can reassure ourselves that this approach is appropriate by examining the limits of Eqs. (14.18) and (14.19) as \( b \) becomes small. In that limit, both \( \tau = bt/m \) and \( \alpha = bv_0/(mg) \) also become small. To assess the limit, we need to expand Eq. (14.18) to second order in \( \tau \) and \( \alpha \), finding that

\[
0 = \tau - (1 + \alpha \sin \theta) \left[ 1 - \left(1 - \tau + \frac{\tau^2}{2}\right)\right] = \tau + (1 + \alpha \sin \theta) \left[-\tau + \frac{\tau^2}{2}\right]
\]

\[
= -\alpha \tau \sin \theta + \frac{\tau^2}{2} = 0 \quad \Rightarrow \quad \tau = 2\alpha \sin \theta
\]

(14.20)

Then, substituting this result into Eq. (14.19), we find that

\[
\frac{R(\theta, \alpha \to 0)}{v_0^2/g} = \frac{\cos \theta}{\alpha(1 + \alpha \sin \theta)} (2\alpha \sin \theta) = \frac{2 \sin \theta \cos \theta}{1 + \alpha \sin \theta} \rightarrow 2 \sin \theta \cos \theta
\]

(14.21)

which is in complete agreement with the known result for the range of a projectile in the absence of air resistance.
14.1.5 Energy Levels in a Quantum Well

The quantum mechanical analysis for the energy levels of a particle of mass $m$ in a one-dimensional quantum well characterized by the potential energy

$$V(x) = \begin{cases} \infty & x < 0 \\ -V_0 & 0 \leq x \leq a \\ 0 & a \leq x \end{cases}$$ (14.22)

leads to the conclusion that the allowed energies $E$ should satisfy the equation\(^5\)

$$s \cot s = -\sqrt{c^2 - s^2}$$ (14.23)

where $c^2 = 2ma^2V_0/h^2$ and $s^2 = c^2(1 - E/V_0)$ or $E = -V_0(1 - s^2/c^2)$. To find the energies, we must thus solve Eq. (14.23) for acceptable values of $s$ once the depth of the well conveyed by the (fixed) value of $c$ has been specified.

A simpler version of this equation emerges if we square it (thereby possibly introducing spurious roots because the squared equation is also consistent with a plus sign on the right hand side) and then invoke the trigonometric identity $1 + \cot^2 s = \sin^{-2} \theta$ to find that

$$s^2 \cot^2 s = s^2 (\sin^{-2} s - 1) = c^2 - s^2 \quad \Rightarrow \quad \sin s = \pm \frac{s}{c}$$ (14.24)

We can be confident that all solutions of Eq. (14.23) will satisfy Eq. (14.24), but we cannot be sure that all solutions of Eq. (14.24)—a potentially larger set—will satisfy Eq. (14.23). Thus, we must in the end remember to sort from all the roots of Eq. (14.24) only those that also satisfy Eq. (14.23).

While we might (see exercises) be interested in the way the allowed energies change as the depth and width of the well change (i.e., as $c$ changes), we shall here illustrate the techniques by supposing a particular well, namely one for which $c = 25$. Then, we seek solutions specifically to

$$\sin s = \pm 0.04s \quad \text{or} \quad \sin s = \mp 0.04s = 0$$ (14.25)

We suppose we seek roots of this function in the interval $s \geq 0$. The graphs in Fig. 14.5 reveal these solutions in several ways. The lower graph in Fig. 14.5(a) shows the function $\sin s - 0.04s$ (upper sign in Eq. (14.25)), and its roots lie where this graph crosses the $s$ axis; there are eight roots in this group. Similarly, the upper graph in Fig. 14.5(a) shows the function $\sin s + 0.04s$ (lower sign in Eq. (14.25)), and its roots appear where the graph crosses the $s$ axis; there are nine roots in this group. In Fig. 14.5(b), which shows the functions $\sin s$, $+0.04s$, and $-0.04s$, the roots appear at the values of $s$ where one or the other of the sloped straight lines intersects the sine curve. From either of these graphs, we conclude that, for the upper sign in Eq. (14.25) the eight roots are bounded—crudely—by

<table>
<thead>
<tr>
<th>root 1u</th>
<th>-0.5 &lt; s &lt; 0.5</th>
<th>root 5u</th>
<th>12.5 &lt; s &lt; 13.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>root 2u</td>
<td>2.5 &lt; s &lt; 3.5</td>
<td>root 6u</td>
<td>14.5 &lt; s &lt; 15.5</td>
</tr>
<tr>
<td>root 3u</td>
<td>6.0 &lt; s &lt; 7.0</td>
<td>root 7u</td>
<td>19.0 &lt; s &lt; 20.0</td>
</tr>
<tr>
<td>root 4u</td>
<td>8.5 &lt; s &lt; 9.5</td>
<td>root 8u</td>
<td>20.5 &lt; s &lt; 21.5</td>
</tr>
</tbody>
</table>

and that, for the lower sign, the nine roots are bounded—again crudely—by

<table>
<thead>
<tr>
<th>root 1l</th>
<th>-0.5 &lt; s &lt; 0.5</th>
<th>root 6l</th>
<th>16.0 &lt; s &lt; 17.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>root 2l</td>
<td>3.0 &lt; s &lt; 4.0</td>
<td>root 7l</td>
<td>17.5 &lt; s &lt; 18.5</td>
</tr>
<tr>
<td>root 3l</td>
<td>5.5 &lt; s &lt; 6.5</td>
<td>root 8l</td>
<td>22.5 &lt; s &lt; 23.5</td>
</tr>
<tr>
<td>root 4l</td>
<td>9.5 &lt; s &lt; 10.5</td>
<td>root 9l</td>
<td>23.5 &lt; s &lt; 24.5</td>
</tr>
<tr>
<td>root 5l</td>
<td>11.5 &lt; s &lt; 12.5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

\(^5\)Text books in quantum mechanics usually treat the finite-depth well for which $V(x) = -V_0$ in $-a \leq x \leq a$ and $V(x) = 0$ outside that interval. Nevertheless, the strategy for deriving the result in Eq. (14.23) is described in almost every intermediate level text on quantum mechanics. See, for example, Section 2.6 in David J. Griffiths, *Introduction to Quantum Mechanics* (Prentice Hall, Inc., Upper Saddle River, NJ, 1995). Actually, the condition we choose for illustration gives results identical to those for the odd states in the more conventional well.
Figure 14.5: Graphs to reveal solutions to Eq. (14.25). In each frame, the graph labeled ‘up’ corresponds to the upper sign in Eq. (14.25) and the graph labeled ‘low’ corresponds to the lower sign in that equation.

14.2 Symbolic Approaches

Symbolic approaches to finding roots always take advantage of specific features of the equation or system of equations for which roots are sought. The roots of at least low order polynomials can be found symbolically. Most of us learned in high school algebra, for example, that the (single) root of a linear polynomial is given by

\[ f(x) = ax + b = 0 \quad \Rightarrow \quad x = \frac{-b}{a} \]  

(14.28)

and that the quadratic formula

\[ f(x) = ax^2 + bx + c = 0 \quad \Rightarrow \quad x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} = \frac{2c}{-b \mp \sqrt{b^2 - 4ac}} \]  

(14.29)

gives the (two) roots of a quadratic polynomial. Similar—though more complicated—expressions exist for the roots of cubic and quartic polynomials. Roots of a polynomial of fifth or higher order cannot be found symbolically unless the polynomial happens to factor into a product of polynomials, each of which individually is of order no higher than quartic. Further, most learned in high school trigonometry that

\[ f(x) = \sin(kx) = 0 \quad \Rightarrow \quad x = \frac{n\pi}{k}, \quad n = 0, \pm 1, \pm 2, \ldots \]  

(14.30)
Occasionally, we will encounter an equation that looks intractable but that can be converted into a tractable case with an appropriate variable transformation. The roots of the expression

\[ f(x) = \frac{a + x}{b + x^2} - c \]  

for example, can be found symbolically because, so long as \( b + x^2 \neq 0 \), the simple recasting

\[ \frac{a + x}{b + x^2} - c = 0 \implies a + x - c(b + x^2) = -cx^2 + x + a - cb = 0 \]  

converts the problem into one involving a quadratic polynomial. Beyond these few cases, very few roots can be found symbolically.

### 14.6 Algorithms for Finding Roots Numerically

In this section we describe several methods for determining a root of a general function numerically.

#### 14.6.1 The Method of Bisection

Suppose, for example, that we know that a single root of \( f(x) \) lies between \( x_{\text{min}} \) and \( x_{\text{max}} \). We can, of course, calculate \( f_{\text{min}} = f(x_{\text{min}}) \) and \( f_{\text{max}} = f(x_{\text{max}}) \). With this input, we can refine our knowledge of the interval in which the root lies by

1. calculating

\[ x_{\text{mid}} = \frac{1}{2} \left( x_{\text{min}} + x_{\text{max}} \right) \quad \text{and} \quad f_{\text{mid}} = f(x_{\text{mid}}) \]  

2. calculating \( f_{\text{min}} f_{\text{mid}} \), which will be positive if \( f_{\text{min}} \) and \( f_{\text{mid}} \) have the same sign (and the root then lies in the upper half of the original interval) and negative if \( f_{\text{min}} \) and \( f_{\text{mid}} \) have opposite signs (and the root then lies in the lower half of the original interval). Note that we have here assumed the root to be a single root (or at least a root of odd multiplicity), since the criterion invoked depends on the function having opposite signs on opposite sides of the root. (For a root of even order, the function has the same sign on opposite sides of the root, and the criterion here described will not identify the half of the interval in which the root lies. Without embellishment, bisection will fail in this case.)

3. refocusing our attention on the interval \( x_{\text{min}} < x < x_{\text{mid}} \) (i.e., replacing \( x_{\text{max}} \) with \( x_{\text{mid}} \)) or on the interval \( x_{\text{mid}} < x < x_{\text{max}} \) (i.e., replacing \( x_{\text{min}} \) with \( x_{\text{mid}} \)), depending on the outcome of the test at step (2).

4. returning to step (1).

The process is illustrated in Fig. 14.6. In the two cases illustrated, the root lies in the upper half of the original interval, so the second iteration will apply the same procedure to the interval from \( x_{\text{mid}} \) to \( x_{\text{max}} \). With each successive iteration, the interval within which we know the root to lie is shrunk to half of its size at the start of that iteration. This method of bisection is guaranteed to converge provided only that there is a root in the interval, though it will not work if the root is a root of even order and it may be confused if the original interval happens to contain several roots. The iteration is continued until the interval has been reduced to a size that we are willing to accept as a tolerance or, though it happens rarely, if \( f_{\text{mid}} \) actually is zero at some point in the process.
14.6.2 Newton’s Method

A second—more efficient but also less stable—algorithm requires a single starting value rather than a pair of values that bracket a root. Suppose that \( x_n \) is the current estimate of the position of the desired root, and let \( f_n = f(x_n) \). In Newton’s method, we calculate the next iterate \( x_{n+1} \) by

1. calculating the derivative of the function at \( x_n \), i.e., \( df(x)/dx|_{x_n} = f'_n \).
2. extrapolating the tangent line to the graph of the function—a line whose slope is \( f'_n \)—to the point at which it intersects the horizontal axis, i.e., finding \( x_{n+1} \) by requiring it to satisfy

\[
\Delta f = 0 - f_n = f'_n (x_{n+1} - x_n) \quad \implies \quad x_{n+1} = x_n - \frac{f_n}{f'_n}
\]

(14.34)

The relevant geometry is illustrated in Fig. 14.7. Given an initial “guess”, Newton’s method will converge more rapidly than the method of bisection, though it will have difficulties if the initial guess—or, for that matter, the root itself—is too close to a point at which the derivative of the function is zero. With a poor initial guess, the method may diverge altogether or may converge on a root remote from the one sought.

14.6.3 Other Methods

Numerous other methods for finding roots, some of them restricted to polynomials, have been developed. Because available routines in some programs sometimes use methods other than the two
described above in detail, we include here a brief outline of the main idea in each of several other methods:

- **The secant method** starts from two approximations to the root, say \( x_1 \) and \( x_2 \), calculates \( f_1 = f(x_1) \) and \( f_2 = f(x_2) \), fits a straight line through the points \( (x_1, f_1) \) and \( (x_2, f_2) \), finds the intersection point \( x_3 \) of that line with the \( x \) axis, and then repeats the process with the two approximations \( x_2 \) and \( x_3 \). The method is similar to Newton’s method, except that it uses the values of the function at two points to estimate the slope of the tangent line to the function \( f(x) \). It does not require explicit knowledge of the derivative of \( f(x) \).

- **Müller’s method** is mildly more sophisticated than the secant method but involves a similar idea. It starts with three estimates of the root, say \( x_1 \), \( x_2 \), and \( x_3 \), calculates \( f_1 = f(x_1) \), \( f_2 = f(x_2) \), and \( f_3 = f(x_3) \), fits a parabola to the points \( (x_1, f_1) \), \( (x_2, f_2) \), and \( (x_3, f_3) \), finds the roots of that parabola using the form of the quadratic formula given in Eq. (14.29), and finally takes as \( x_4 \)—which replaces \( x_3 \) for the next iteration—the one root that emerges when the ambiguous sign in Eq. (14.29) is chosen so that the denominator has the larger of the two possible absolute values.

- **Laguerre’s method**, which is limited to polynomials, (1) recognizes that, in terms of its roots \( x_i \), a polynomial scaled so that the coefficient of its highest order term is 1 (which has no effect on its roots) can be expressed in the form

\[
f(x) = \prod_{i} (x - x_i)
\]

(14.35)

(2) finds a couple of relationships satisfied by polynomials in this form, (3) assumes with only weak justification a priori that the root sought is isolated from all the others (which are clustered together), and (4) deduces that the next iterate \( x_{n+1} \) for the root sought should be determined from the current guess \( x_n \) by

\[
x_{n+1} = x_n - \frac{N}{G \pm \sqrt{(N - 1)(NH - G^2)}}
\]

(14.36)

where the sign in the denominator is chosen so that the denominator has the larger of the two possible absolute values, \( N \) is the order of the polynomial (equal to the number of roots), and \( G \) and \( H \) are defined by \( G = f'f / f \) and \( H = (f'f)^2 - f''f, \) each evaluated at \( x = x_n \).

### 14.6.4 Assessing Accuracy

As with all numerical operations, assessing the accuracy of the roots found by an algorithm is essential before we can have confidence in the roots. We can, of course, always substitute the root we have found into the function whose root we seek and simply notice how close to zero the value of the function at the proposed root actually is. In the end, that comparison may provide the most important test of accuracy, though whether the value \( f(x_{\text{root}}) \approx 0.0001 \), say, is “close enough” to zero requires a judgment that would take into account the magnitude of \( f(x) \) over the important domain of \( x \) and, even more, the magnitude of the derivative of \( f(x) \) evaluated at the root.\(^{6}\)

Certainly, that criterion provides the most accessible test we might apply. It is also a criterion that is easily implemented in a computer program that monitors convergence and stops automatically when the absolute value of the function has been reduced below some specified tolerance.

We would, however, be more interested in the accuracy of the root itself, i.e., in the amount \( |x_{\text{exact}} - x_{\text{approx}}| \) by which the (unknown) exact root differs from the root on which a particular root

\(^{6}\) The less steeply sloped the function at the root, the harder it is to obtain an accurate root by using the value of the function as a guide.
algorithm converges. Only the method of bisection supports a clear assessment of that difference
since, at any particular step in the progress of that algorithm towards a root, we know that the root is
“trapped” between two values whose separation is halved with each step in the iteration. Continuing
until the difference $|x_{i+1} - x_i|$—sometimes called the residual—between consecutive iterates is less
than some specified tolerance guarantees that we have located the root to within that tolerance.
Implemented in a computer program, an algorithm using the method of bisection can monitor the
separation of consecutive iterates and stop when that separation, either as an absolute value or as a
fraction of its current value, is reduced below a specified absolute or relative tolerance, respectively.

Given appropriate initial bounds, the method of bisection is guaranteed to converge on a root,
but the rate of convergence is slow by comparison with other methods. Assessing the accuracy
of the root itself for those other algorithms is more difficult. If, however, the convergence of the
algorithm in use is fairly rapid (and Newton’s method will usually satisfy this expectation), taking
the (absolute) accuracy of a particular iterate to be on the order of the difference between it and
the next iterate, i.e., on the order of the residual, is reasonable. For these algorithms, as with the
method of bisection, a computer implementation of a method in the category of this paragraph
can also monitor the residual and stop when it has been reduced below a specified (absolute or
relative) tolerance. For these methods, however, convergence is not guaranteed, and computer
implementations of these methods should include an alternate criterion for stopping the iteration.
Such an implementation might, for example, limit the number of iterations and—to keep the user
fully informed—display a warning message if the iteration is stopped because this limit is exceeded
rather than because the residual has been made sufficiently small. Without some such fail-safe
stopping criterion, these methods are in danger of iterating forever.

Beyond monitoring the value of the function or the residual, seeking a particular root by more
than one method may give some insight into the accuracy of that root. Different methods have
different strengths, weaknesses, and quirks. When they agree to some number of digits, we can have
more confidence in the result than we would have if we had obtained it by only one method. When
they disagree, we have at least a hint that the function at hand possesses some pathology that we
should perhaps attempt to understand before accepting the root we have found.

14.15 Finding Roots Numerically with C

C programs to find roots numerically can be constructed in several ways. In this section, we describe
how we can implement one or another algorithm directly in a program that we write from scratch—a
task that is tedious and difficult except for the simplest of algorithms. In addition, we describe how
we can make use of available standard subroutines—we here focus on those in the Numerical Recipes
library—and devote our efforts solely to the easier task of writing a driving program to invoke the
features of whatever root-finding subroutine we choose to use.

14.15.1 C Programs from Scratch

Note: All C programs (*.c) and all C-generated data files (*.c.dat) in this subsection can be
copied from the directory $HEAD/cc. $HEAD for your site is defined in the Local Guide.

The very simplest method for finding a root is to keep the person desiring the root in the loop
and write a program that asks that person for a proposed root, then calculates and displays the value
of the function at the proposed root, and finally returns to ask for a new proposal. For the polynomial
of Eq. (14.3), whose roots give the turning points for a classical particle experiencing a particular
potential energy, we might, for example, use the program listed in Table 14.1. After storing this
Table 14.1: The C program guessroot.c.

```c
/* PROGRAM guessroot.c */

#include <stdio.h> /* Load standard i.o routines */
#include <math.h> /* Load standard math routines */

main()
{
  float x, f; /* For trial root, function */
  printf("First guess : "); scanf("%f", &x);
  while( x != 9999 ){
    f = pow(x,3)/10000.0 + pow(x,2)/200.0 - x/500.0 - 0.5;
    printf("f(x) = %15.6f
", f);
    printf("Next guess : "); scanf("%f", &x);
  }
}
```

program in the file guessroot.c, we compile, link, and run it with the two statements

```
cc -o guessroot.xc guessroot.c -lm
./guessroot.xc
```

We know for the illustrative function used in the above program that there is a root in the vicinity of \( x = -50.0 \), the lowest turning point for the potential energy of Eq. (14.3). To find that root, we might then use this program to evaluate the function repeatedly for a succession of judicious guesses, each guided by the results of all previous trials. We might, for example, find this root with the “conversation”

```
./guessroot.xc
First guess : -50.0 f(x) = -0.400000
Next guess : -49.0 f(x) = -0.161900
Next guess : -48.0 f(x) = 0.056800
Next guess : -48.5 f(x) = -0.050163
Next guess : -48.25 f(x) = 0.003911
Next guess : -48.25 f(x) = -0.022977
Next guess : -48.2813 f(x) = -0.002794
Next guess : -48.2656 f(x) = 0.000572
Next guess : -48.2676 f(x) = -0.000264
Next guess : -48.2676 f(x) = 0.000143
```

---

7To be specific, both here and throughout this section, we illustrate with the statements that would be used in UNIX to compile, link, and run the program. Other operating systems probably accomplish the same end with different statements. In particular, preceding the program name with ./ is necessary in UNIX but may not be necessary with other operating systems. Details will be found in the Local Guide.

8The final argument -lm is needed so that compiler will know to include the C math libraries, some pieces of which are used by this program.

9To be specific, both here and throughout this section, we illustrate these steps with the statements that would be used in UNIX to compile, link, and run the program. Other operating systems probably accomplish the same end with different statements. In particular, preceding the program name with ./ is necessary in UNIX but may not be necessary with other operating systems. Details will be found in the Local Guide.

10The prompt, the entered values, and the returned values have been displayed on the same line to save space.
CHAPTER 14. FINDING ROOTS

Next guess : -48.2685  f(x) = -0.000050
Next guess : 9999

Starting at a point we suspect to be below the root, we scan upwards until the function changes sign. In that way, we discover that the root lies between \( x = -49.0 \) and \( x = -48.0 \). From this point, we carry out a binary search, repeatedly halving the interval and deciding on the basis of the signs at each step which half of the current interval to refine further. By the time we stopped, we had concluded that the root lies somewhere between \( x = -48.2676 \) and \( x = -48.2685 \), i.e., that the root is \( x = -48.268 \) to three places after the decimal.

An algorithm to implement the method of bisection in a C program merely automates the decision making process that was explicit in the previous paragraph. A full listing of a possible program, which we have named \texttt{bisect.c}, is presented in Section 14.E. Beyond incorporating the necessary header files, this program (1) defines the function whose roots will be sought with the lines

```c
float func( float x )
{
    return pow(x,3)/10000.0 + pow(x,2)/200.0 - x/500.0 - 0.5;
}
```

(2) declares floating variables \( \texttt{tol, xa, xm, xb, fa, fm, and fb} \) for the tolerance, the lower, middle, and upper values of the interval, and the values of the function at \( \texttt{xa, xm, xb} \); (3) sets the tolerance and reads the initial bounds for the root with the lines

```c
tol = 0.0001
printf( "\nLower bound = " ); scanf( "%f", &xa );
printf( "\nUpper bound = " ); scanf( "%f", &xb );
```

and finally (4) evaluates the function at the two bounds, verifies that a root exists in the specified interval (and terminates execution if not), refines the estimate of the root, and displays the final value of the root, its final bounds, and the function at the declared root with the lines

```c
fa = func(xa); fb = func(xb); fm = fa;
if (fa*fb >= 0.0) { printf( "No root in interval\n" ); return; };
while( (xb-xa) >= tol & fm != 0.0 ){xm = 0.5*(xb+xa);
    fm = func(xm);
    if (fa*fm >= 0.0) xa = xm; else xb = xm;
}
printf( "root = %15.6f\n", xm );
printf( "xb - xa = %15.6f\n", xb-xa );
printf( "function = %15.6f\n", func( xm ) );
```

(We must, of course, remember to declare the main program \texttt{void} in order to get away with the simple statement \texttt{return} to terminate execution and return to the operating system.) The loop continues as long as the interval within which the root lies is larger than the desired tolerance and the value at the midpoint hasn’t happened to be \textit{exactly} zero. Using this program, which we compile, link, and run with the statements

```
cmake -o bisect.xc bisect.c -lm
./bisect.xc
```

we have the conversation
14.15. FINDING ROOTS NUMERICALLY WITH C

./bisect.xc
Lower bound = -50.0
Upper bound = -40.0
root = -48.268204
xb - xa = 0.000076
function = 0.000014

and find that the lowest root of our sample polynomial is \( x = -48.2682 \pm 0.0001 \). If we instead use the lower and upper bounds \(-15.0\) and \(-5.0\), this program converges to \( x = -11.0804 \pm 0.0001 \). Finally, with lower and upper bounds 5.0 and 15.0, the program converges to \( x = 9.3487 \pm 0.0001 \).

14.15.2 Using Numerical Recipes

**Note:** CPSUP-generated C programs (*.c) and C-generated data files (*.c.dat) can be downloaded from the directory $HEAD/cc. If your site has the appropriate license, files in this subsection from the Numerical Recipes library can be downloaded from a subdirectory of the directory $NRHEAD (see Fig. 10.1). $NRHEAD and $HEAD for your site are defined in the Local Guide.

Numerical Recipes provides several subroutines for finding roots of user-defined functions of a single variable, including rtbis (bisection), rtnewt (Newton’s method), and rtsafe (combination of bisection and Newton’s method to achieve greater stability). For the sake of a specific example, suppose we seek non-negative values of \( s \) satisfying Eq. (14.25), i.e., we seek energy eigenvalues for the quantum particle in a finite depth well. The graph of this function in Fig. 14.5 supported the identification of the brackets shown in Eqs. (14.26) and (14.27) for the non-negative roots of this function.

The *method of bisection* is implemented in the function rtbis.c, which is invoked with a statement of the form

```c
root = rtbis( func, slb, sub, eps )
```

where `func` defines the univariate function whose roots are to be found, `slb` and `sub` specify the bounds within which the desired root is known to lie, and `eps` specifies the desired absolute accuracy.

A C driving program using `rtbis` must be written by the user and will have the general form shown in Table 14.2. This program incorporates necessary include files and headers, defines the function whose roots are desired, establishes the bounds and the desired error for a given root, invokes `rtbis`, and displays the value obtained. The steps in finding this root then are

1. Create the driving program, either (1) by copying the demonstration program xrtbis.c to the default directory from the directory\(^{11}\) $NRHEAD/recipes_c-ansi/demo/src, giving it the name sinrtbis.c, and editing it appropriately or (2) by creating the file sinrtbis.c with an available text editor.

2. Copy the file rtbis.c from the directory $NRHEAD/recipes_c-ansi/recipes to the default directory.

3. Copy the header files nr.h and nrutil.h to the default directory from the directory $NRHEAD/recipes_c-ansi/include.

4. Copy the utility subroutine nrutil.c to the default directory from the directory $NRHEAD/recipes_c-ansi/recipes.

5. Compile and link the executable module with the statement

\(^{11}\)Alternatively, at some sites, the language directory may be recipes_c-kr rather than recipes_c-ansi.
Table 14.2: The C program sinrtbis.c. A more elaborate demonstration program named xrtbis.c can be found in the directory $NRHEAD/recipes_c-ansi/demo/src.

```c
/* PROGRAM sinrtbis.c */
#include <stdio.h>  /* Load standard i/o routines */
#include <math.h>   /* Load standard math routines */
#include "nr.h"     /* Load necessary recipes headers */
#include "nrutil.h"

float func( float s ) /* Define function */
{
  return sin(s) - 0.04*s;
}

main()
{
  float eps, slb, sub;  /* For tolerance, for bounds */
  float root;           /* For root */

  printf( "Lower bound on root = " ); /* Get lower bound */
  scanf( "%f", &slb );
  printf( "Upper bound on root = " ); /* Get upper bound */
  scanf( "%f", &sub);
  printf( "Error limit = " ); /* Get error limit */
  scanf( "%f", &eps);

  root = rtbis( func, slb, sub, eps ); /* Find root */

  printf( "Root is %15.6f\n", root );
}
```

```
c -o sinrtbis.xc sinrtbis.c rtbis.c nrutil.c -lm
```

6. Run the program with the statement ./sinrtbis.xc.

The lowest two and the highest non-negative roots of the sample function would then be found with the statements\textsuperscript{12}

```
./sinrtbis.xc
Lower bound on root = -0.5
Upper bound on root = 0.5
Error limit = 0.0001
Root is 0.000000
```

\textsuperscript{12}To save space, prompting messages and values input are shown on the same line.
14.15. FINDING ROOTS NUMERICALLY WITH C

./sinrtbis.xc
Lower bound on root = 2.5
Upper bound on root = 3.5
Error limit = 0.0001
Root is 3.020508

./sinrtbis.xf
Lower bound on root = 20.5
Upper bound on root = 21.5
Error limit = 0.0001
Root is 20.994324

An alternative algorithm uses Newton’s method and is implemented in rtnewt.c. While the method of bisection requires an upper and lower bound in order to locate a root, Newton’s method starts with a single zeroth-order “guess”. Even so, rtnewt still requires bounds for each root. The bounds are used to determine the necessary initial guess—a value at the midpoint of the bounded interval—and, since Newton’s method has some instabilities, they are also used as a way to assess whether Newton’s method is stepping out of the interval in which a root is known to exist. The function rtnewt is invoked with a statement of the form

```
root=rtnewt( func, slb, sub, eps );
```

where func defines a procedure that returns both the function whose roots are to be found and its first derivative, slb and sub specify the bounds within which the desired root is known to lie, and eps specifies the desired absolute accuracy. A suitable driving program to use rtnewt to find the roots of our test function differs from sinrtbis in only a few respects:

- The first line would name the program sinrtnewt.
- The line invoking rtbis should invoke rtnewt instead.
- The user-defined function must be replaced with a procedure that returns both the value of the function and the value of its derivative. Thus, the function func in sinrtbis must be replaced with the procedure

```
/* Define function and derivative */

void func( float s, float *f, float *dfdx )
{
    *f = sin(s) - 0.04*s;
    *dfdx = cos(s) - 0.04;
}
```

The steps in finding this root then are

1. Create the driving program, either (1) by copying the demonstration program xrtnewt.c to the default directory from the directory $NRHEAD/recipes_c-ansi/demo/src, giving it the name sinrtnewt.c, and editing it appropriately or (2) by creating the file sinrtnewt.c with an available text editor.

2. Copy the file rtnewt.c from the directory $NRHEAD/recipes_c-ansi/recipes to the default directory.

3. If they aren’t still in the default directories, copy the header files nr.h and nrutil.h and the utility subroutine nrutil.c to the default directory as for rtbis.c.
4. Compile and link the executable module with the statement

```
c -o sinrtnewt.xc sinrtnewt.c rtnewt.c nrutil.c -lm
```

5. Run the program with the statement `.sinrtnewt.xc`.

The lowest two and the highest non-negative roots of the sample function would then be found with the statements\(^{13}\)

```
./sinrtnewt.xc
Lower bound on root = -0.5
Upper bound on root = 0.5
Error limit = 0.0001
Root is 0.000000
```

```
./sinrtnewt.xc
Lower bound on root = 2.5
Upper bound on root = 3.5
Error limit = 0.0001
Root is 3.020478
```

```
./sinrtnewt.xc
Lower bound on root: 20.5
Upper bound on root: 21.5
Error limit: 0.0001
Root is 20.994286
```

As we have already mentioned, Newton's method converges quite rapidly when it works but the price paid for that more rapid convergence is occasional instability. The *Numerical Recipe rtsafe* finds roots by a combination of Newton's method and the method of bisection, thereby achieving an element of stability that Newton's method alone lacks. Exploration of this alternative to *rtbis* and *rtnewt* is left to the exercises at the end of the chapter.

Finally, we mention an auxiliary subroutine *zbrak*, whose function is to find bracketing intervals for any roots in a given interval. This routine is valuable in helping to find the positions of roots prior to invoking *rtbis*. It is called with the statement

```
zbrak(func, a, b, n, slb, sub, &nroots)
```

When called, *zbrak* divides the interval \(a \leq x \leq b\) into \(n\) equal segments, scans through those segments, identifies any in which the function \(func\) changes sign, and stores the lower and upper bounds of each such segment in \(slb[]\) and \(sub[]\). On input \(nroots\) specifies the maximum number of such segments that can be stored; on return, \(nroots\) stores the actual number of such segments found in the specified interval. A suitable driving program must dimension \(slb\) and \(sub\) appropriately, invoke *zbrak*, and then invoke *rtbis* in a loop whose index steps from one bracket to the next. The demonstration programs `$NRHEAD/recipes_c-ansi/demo/src/xrtbis.c, .../xrtnewt.c, and .../xrtsafe.c` all illustrate this strategy. Note that *zbrak* will—erroneously—conclude that a root exists at a point on one side of which the function diverges to \(+\infty\) and on the other side of which the function diverges to \(-\infty\), so we must be cautious in using *zbrak* on functions that have singularities in the interval in which roots are sought.

\(^{13}\)To save space, the prompting messages and the responses are shown on the same line.
14.16 Solving Simultaneous Equations

To this point in this chapter, we have limited ourselves to solving a single equation for a single unknown quantity—though the single equation has frequently exhibited numerous distinct roots. In many contexts, however, the task of finding (one or more) roots will require solving a system of \( n \) simultaneous equations for \( n \) unknowns.\(^{14}\) As with a single equation determining a single unknown, a system of \( n \) equations determining \( n \) unknowns may exhibit more than one solution, each consisting of \( n \) values, one for each of the unknowns. In this section, we merely enumerate a few contexts in which systems of equations arise and outline the strategies for addressing their solution. Fuller discussion can be found in any number of books on linear algebra and/or numerical methods.\(^{15}\)

14.16.1 Systems of Linear Equations

By far the simplest situation occurs when the equations in the system are all linear, i.e., when the unknowns in the system occur only to the first power and never in product with one another. As with a single equation determining a single unknown, a system of \( n \) equations determines \( n \) unknowns, each consisting of \( n \) values, one for each of the unknowns. In this section, we merely enumerate a few contexts in which systems of equations arise and outline the strategies for addressing their solution. Fuller discussion can be found in any number of books on linear algebra and/or numerical methods.

Whatever their physical origin, systems of linear equations can most conveniently be presented in matrix form, e.g.,

\[
\begin{pmatrix}
  a_{11} & a_{12} & a_{13} \\
  a_{21} & a_{22} & a_{23} \\
  a_{31} & a_{32} & a_{33}
\end{pmatrix}
\begin{pmatrix}
  x_1 \\
  x_2 \\
  x_3
\end{pmatrix}
= 
\begin{pmatrix}
  b_1 \\
  b_2 \\
  b_3
\end{pmatrix}
\]

(14.37)

where, with \( i \) and \( j \) independently assuming the values 1, 2, and 3, the quantities \( x_i \) are the unknowns, the quantities \( b_i \) are the inhomogenieties, and the quantities \( a_{ij} \) are the coefficients defining the equations. Here, we have three equations and three unknowns. More generally, for a system of \( n \) equations and \( n \) unknowns, we would have an \( n \times n \) matrix \( A \) of coefficients, an \( n \) element vector \( \mathbf{x} \) of unknowns and an \( n \) element vector \( \mathbf{b} \) of inhomogenieties, we would write the equations compactly in the form

\[
A \mathbf{x} = \mathbf{b}
\]

(14.38)

and we would write their solution formally as

\[
\mathbf{x} = A^{-1} \mathbf{b}
\]

(14.39)

Less compactly but more usefully (at least occasionally), we might remember Cramer’s rule and write the solution in terms of determinants in the form

\[
x_1 = \frac{\begin{vmatrix}
  b_1 & a_{12} & a_{13} \\
  b_2 & a_{22} & a_{23} \\
  b_3 & a_{32} & a_{33}
\end{vmatrix}}{\begin{vmatrix}
  a_{11} & a_{12} & a_{13} \\
  a_{21} & a_{22} & a_{23} \\
  a_{31} & a_{32} & a_{33}
\end{vmatrix}} ;
\quad
x_2 = \frac{\begin{vmatrix}
  a_{11} & b_1 & a_{13} \\
  a_{21} & a_{22} & a_{23} \\
  a_{31} & a_{32} & a_{33}
\end{vmatrix}}{\begin{vmatrix}
  a_{11} & a_{12} & a_{13} \\
  a_{21} & a_{22} & a_{23} \\
  a_{31} & a_{32} & a_{33}
\end{vmatrix}} ;
\quad
x_3 = \frac{\begin{vmatrix}
  a_{11} & a_{12} & b_1 \\
  a_{21} & a_{22} & a_{23} \\
  a_{31} & a_{32} & a_{33}
\end{vmatrix}}{\begin{vmatrix}
  a_{11} & a_{12} & a_{13} \\
  a_{21} & a_{22} & a_{23} \\
  a_{31} & a_{32} & a_{33}
\end{vmatrix}}
\]

(14.40)

In this rule, which can be readily extended to \( n \) equations, the denominator in the expression for the \( i \)-th unknown is the determinant of the coefficient matrix and the numerator is the determinant of the matrix created from the coefficient matrix by replacing its \( i \)-th column with the column of

\(^{14}\) Actually, systems will sometimes be underdetermined (\( n \) equations with \( m \) unknowns, \( n < m \)) or overdetermined (\( n \) equations with \( m \) unknowns, \( n > m \)), but we shall not consider these cases at all.

inhomogeneities. Cramer’s rule provides a direct, symbolic, and exact solution to the system of \( n \) simultaneous linear equations. Note, however, that the rule gives problems if the determinant of the coefficient matrix happens to be zero or, equivalently, if the inverse \( A^{-1} \) of the coefficient matrix fails to exist. When \( |A| = 0 \), the equations are said to be singular and will either have no solution (equations inhomogeneous; \( \mathbf{b} \neq 0 \)) or an infinite number of solutions (equations homogeneous; \( \mathbf{b} = 0 \)).

While compact, Cramer’s rule is not particularly useful for numerical solution of even modest sized systems, since the most direct approach to evaluating determinants is vulnerable to roundoff error.\(^{16}\) We can, however, invent alternative methods that are computationally more satisfactory. The simplest algorithm to describe involves Gaussian elimination, in which one variable at a time is systematically eliminated to yield a simpler system whose solution is readily found by a process called backsubstitution. We illustrate with Eq. (14.37), but the schema is readily extended to \( n \) equations. The process of Gaussian elimination entails

- dividing each equation by the coefficient of \( x_1 \), obtaining\(^ {17}\)

\[
\begin{pmatrix}
1 & a'_{12} & a'_{13} \\
1 & a'_{22} & a'_{23} \\
1 & a'_{32} & a'_{33}
\end{pmatrix}
\begin{pmatrix}
x_1 \\
x_2 \\
x_3
\end{pmatrix}
=
\begin{pmatrix}
b'_{1} \\
b'_{2} \\
b'_{3}
\end{pmatrix}
\quad (14.41)
\]

- keeping the first equation and replacing the second and third with the result of subtracting the first from each in turn, obtaining

\[
\begin{pmatrix}
1 & a'_{12} & a'_{13} \\
0 & a''_{22} & a''_{23} \\
0 & a''_{32} & a''_{33}
\end{pmatrix}
\begin{pmatrix}
x_1 \\
x_2 \\
x_3
\end{pmatrix}
=
\begin{pmatrix}
b'_1 \\
b'_2 \\
b'_3
\end{pmatrix}
\quad (14.42)
\]

- dividing the second and third equations by the coefficient of \( x_2 \), obtaining

\[
\begin{pmatrix}
1 & a''_{12} & a''_{13} \\
0 & 1 & a''_{23} \\
0 & 1 & a''_{33}
\end{pmatrix}
\begin{pmatrix}
x_1 \\
x_2 \\
x_3
\end{pmatrix}
=
\begin{pmatrix}
b''_1 \\
b''_2 \\
b''_3
\end{pmatrix}
\quad (14.43)
\]

- keeping the first and second equations but replacing the third with the result of subtracting the second from the third, obtaining

\[
\begin{pmatrix}
1 & a'_{12} & a'_{13} \\
0 & 1 & a''_{23} \\
0 & 0 & a'''_{33}
\end{pmatrix}
\begin{pmatrix}
x_1 \\
x_2 \\
x_3
\end{pmatrix}
=
\begin{pmatrix}
b'_1 \\
b'_2 \\
b'''_3
\end{pmatrix}
\quad (14.44)
\]

In essence, Gaussian elimination converts the original system of equations with a general coefficient matrix into an equivalent system whose coefficient matrix is upper triangular. In that form, however, the solution is readily obtained by backsubstitution. The third equation tells us directly that \( x_3 = b'''_3 / a'''_{33} \). Then, knowing \( x_3 \), we find from the second equation that \( x_2 = b''_2 - a''_{23} x_3 \) and, knowing \( x_3 \) and \( x_2 \), we find from the first equation that \( x_1 = b'_1 - a'_{12} x_2 - a'_{13} x_3 \). The job is done!

Unfortunately, in a computer whose arithmetic is done to finite precision, the order in which the equations are treated in this process and the order in which the variables are placed can have a significant impact on the quality of the solution obtained. Thus, while Gaussian elimination

\(^{16}\)The most direct approach involves sums and differences of products of \( n \) elements taken so that each column and each row is represented once and only once in each product. The result is a combination of terms that are individually large, some of which are positive and some negative. We end up trying to evaluate a difference between two large numbers, an operation that invites roundoff error.

\(^{17}\)For the sake of simplicity, we will not bother to keep track of the relationship between the original coefficients and those generated along the way. The nature of the algorithm will be clear even without that knowledge.
with backsubstitution provides a starting point, it requires sophisticated embellishment to choose
the optimum equation and variable to be the focus of each step in the process. Effecting that
embellishment entails a process called pivoting, in which at each step we examine the coefficients in
the remaining equations and reorder either the equations (partial pivoting) or the equations and
the variables (full pivoting) to optimize the accuracy of the solution. Gaussian elimination with pivoting
(alternatively called Gaussian elimination with pivotal condensation) yields a more involved program
but also increases the likelihood of useful results.

A similar strategy exploits the property that, under appropriate (and not too restrictive) con-
ditions, the coefficient matrix $A$ can be factored into a product of two matrices $L$ and $U$, the first of
which has non-zero elements only on and below the main diagonal (and the elements on the main
diagonal are all ones) and the second of which has non-zero elements only on and above the main
diagonal, and the two matrices are unique. That is, we can write

$$A = LU$$

where

$$L = \begin{pmatrix}
1 & 0 & 0 \\
l_{21} & 1 & 0 \\
l_{31} & l_{32} & 1
\end{pmatrix} \quad \text{and} \quad U = \begin{pmatrix}
u_{11} & u_{12} & u_{13} \\
0 & u_{22} & u_{23} \\
0 & 0 & u_{33}
\end{pmatrix} \quad \text{(14.45)}$$

This so-called $LU$ decomposition allows us to seek the solution of the original equation in steps.
First, we view the equation in the form

$$Ax = b \quad \Rightarrow \quad LUx = b \quad \Rightarrow \quad Ly = b \quad \text{where} \quad y = Ux \quad \text{(14.46)}$$

Since the first equation in the (matrix) equation $Ly = b$ tells us $y_1$ directly, the second tells us $y_2$
directly once $y_1$ is known, and the third tells us $y_3$ directly once $y_1$ and $y_2$ are known, the equation
$Ly = b$ is readily solved for $y$. Then, however, once $y$ is known, a similar process that starts with
$x_3$, then moves to $x_2$ and finally to $x_1$ directly solves the equation $Ux = y$ for the original unknowns in $x$.

The methods outlined in the previous paragraphs will all work for (almost) any system of linear
equations. Sometimes, however, the coefficient matrix may have special properties that can be
exploited to simplify the algorithm or—important for large systems—reduce the requirements on
memory for storage of matrices and intermediate results. The coefficient matrix may be symmetric
($a_{ij} = a_{ji}$) or it may be sparse (only a small fraction of its elements differing from zero). In the
latter category, the matrix may be tridiagonal (non-zero elements only on the main diagonal, on the
diagonal immediately above the main diagonal, and on the diagonal immediately below the main
diagonal). Special savings in storage can be achieved if the matrix happens to be both symmetric
and tridiagonal, since in that case the only elements that need be stored are those on the main
diagonal and those on the diagonal immediately above the main diagonal, $2n - 1$ elements rather
than $n^2$ elements.

The methods outlined in the previous paragraphs are also all direct methods, i.e., each leads
directly to the desired solution in a finite number of steps. Except for roundoff error, each would
yield an exact solution to the equations at hand. When the coefficient matrix is sparse, an iterative
approach may be computationally more efficient. Such approaches entail finding a means by which
an initial guess can be repeatedly refined until some criterion of convergence has been met. Among
the most common such procedures applies to Laplace’s equation. As judiciously as possible, we
“guess” a solution at a regular grid of points laid over the domain of the problem. Then, to carry
out the first iteration, we step systematically through that grid, replacing the value at each grid
point with the average of the values at its nearest neighbors. Then, we repeat the process with the
results of the first iteration as input, generating the second iteration, and continuing until—say—no
value changes by more than some specified tolerance. The only drawback to an iterative method is

---

18The proof of this assertion is beyond the scope of this book. See, for example, S. D. Conte, Elementary Numerical
Analysis (McGraw-Hill, New York, 1965), pg. 178ff, or William H. Press, Brian P. Flannery, Saul A. Teukolsky, and
2.3 for a proof by construction.
that we have added the uncertainties associated with convergence to those potentially generated by computer roundoff. In terms of computational labor, however, the sacrifice is often worth the gains.

Finally, we merely mention the more sophisticated approach—singular value decomposition—we must adopt if the system of equations confronting us happens to be nearly singular, in which case the presence of computer roundoff introduces instabilities in the simpler methods.\footnote{For a full description of this technique, see William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling, \textit{Numerical Recipes} (Cambridge University Press, Cambridge, 1992), Second Edition, Section 2.6.}

Each of the software packages described in this book makes available a spectrum of commands to solve simultaneous linear equations. We here merely enumerate those commands and indicate the general strategy each implements. We leave the descriptive details to the vendor-supplied documentation, and we leave illustrative applications to the exercises at the end of the chapter.

**NUMERICAL RECIPES in C:** For numerical solution of systems of linear equations, the C subroutines in the \textit{Numerical Recipes} library include \texttt{gaussj.c}, which uses Gauss-Jordan elimination (similar to Gaussian elimination); \texttt{ludcmp.c} and \texttt{lubksb.c}, which effect the LU decomposition of the coefficient matrix and the solution of the associated system; \texttt{tridag.c}, which is especially adapted to tridiagonal systems; and \texttt{svdcmp.c} and \texttt{svbksb.c}, which effect the singular value decomposition of the coefficient matrix and the solution of the associated system. All of these routines use direct methods.

### 14.16.2 Systems of Nonlinear Equations

When one or more of the equations in a system to be solved is nonlinear, the task is much more difficult. Occasionally, systematic elimination of one variable at a time, followed by back-substitution, will yield a solution. More often, the system is not amenable to such a simple approach. We must resort to more involved approaches. Fitting experimental data by the method of least squares to functions in which the parameters do not appear linearly and finding the points of equilibrium in a system with more than one independent variable are common sources of such problems.

Since numerical methods for finding the root (or roots) of a nonlinear system of equations are all iterative, possession of a good starting guess is imperative. In two dimensions, where the equations to be solved are

\[
\begin{align*}
  f_1(x_1, x_2) &= 0 \quad ; \quad f_2(x_1, x_2) = 0 \\
  f_1(x_1, x_2) &= \sin\left(\frac{x_1^2}{20}\right) - \cos\left(\frac{x_2}{5}\right) \quad ; \quad f_2(x_1, x_2) = x_1 \tanh(x_2) - 5.0
\end{align*}
\]  

(14.47)

we might begin by drawing a map in the \((x_1, x_2)\) plane showing the zero contours of each function. The map in Fig. 14.8, for example, shows the zero contours for the two functions

\[
\begin{align*}
  f_1(x_1, x_2) &= \sin\left(\frac{x_1^2}{20}\right) - \cos\left(\frac{x_2}{5}\right) \quad ; \quad f_2(x_1, x_2) = x_1 \tanh(x_2) - 5.0
\end{align*}
\]  

(14.48)

The actual intersections of the dashed and solid curves near the labels A reveal two roots. In addition, the dashed and solid curves pass close to one another—but do not actually intersect—in the vicinity of the labels B. The roots near A can probably be found relatively easily by an iterative search procedure. That there are “almost” roots near B may confuse some algorithms and, if those points are close enough to the real roots near A, they might even cause difficulties in finding the real roots.

As the number of independent variables increases, the search described in the previous paragraph would move from intersection points of curves in a plane to intersection points of three surfaces in three-space to intersection points of four hypersurfaces in four-space to \ldots. Sometimes it may be possible (and wise) to solve for some of the variables in terms of the others and temporarily eliminate some variables (i.e., reduce the dimensionality of the search). The task is complicated and, beyond the simple suggestion of striving to reduce the dimensionality, no general guidelines can be given. Any means, however devious, that can be exploited to give clues as to the existence of roots and,
14.16. SOLVING SIMULTANEOUS EQUATIONS

Figure 14.8: Zero contours for the two functions in Eq. (14.48). The function \( f_1(x_1, x_2) \) is shown with solid lines; \( f_2(x_1, x_2) \) is shown with dashed lines. The dashed curve actually intersects the solid curve in two points near the labels A; the two curves almost intersect near the labels B.

even better, to their whereabouts should be exploited as fully as possible before actually embarking on a numerical search.

Once a root has been located approximately, we might adopt a brute force technique, writing a program that

1. Accepts a guessed solution, one value for each unknown,
2. Evaluates the functions and displays the result, and
3. Returns to step 1 for a new guess.

On first pass, we enter the initial guess. Then, after seeing how well that guess works, we make a second (informed or, maybe, random) guess, repeating the process until the values of all functions have been reduced to acceptably small levels. Depending on the dimensionality of the search, we will usually develop a feel for the effect of changes in each member of the guessed solution. Fairly quickly, we may develop a skill at narrowing in on an acceptable solution.

More systematic searches in multi-dimensional parameter spaces are harder to design. One route in particular expands Newton’s method. Suppose, to be specific, we seek solutions to the three nonlinear equations

\[
\begin{align*}
  f_1(x_1, x_2, x_3) &= 0 ; \\
  f_2(x_1, x_2, x_3) &= 0 ; \\
  f_3(x_1, x_2, x_3) &= 0 
\end{align*}
\]  

Suppose, further, that we have examined the equations and determined that there does indeed exist a root in the immediate vicinity of the point \((x_1^{(0)}, x_2^{(0)}, x_3^{(0)})\). We might then suppose that the actual root differs from this guess by a small amount, say,

\[
\begin{align*}
  x_1 &= x_1^{(0)} + \delta x_1 ; \\
  x_2 &= x_2^{(0)} + \delta x_2 ; \\
  x_3 &= x_3^{(0)} + \delta x_3
\end{align*}
\]
and demand that the “corrections” satisfy the equations

\[
0 = f_1(x_1^{(0)} + \delta x_1, x_2^{(0)} + \delta x_2, x_3^{(0)} + \delta x_3) \\
0 = f_2(x_1^{(0)} + \delta x_1, x_2^{(0)} + \delta x_2, x_3^{(0)} + \delta x_3) \\
0 = f_3(x_1^{(0)} + \delta x_1, x_2^{(0)} + \delta x_2, x_3^{(0)} + \delta x_3)
\]  

(14.51)

These equations are, of course, not really any more tractable as they stand than were the original equations. Because the corrections are all presumed small, however, we should be able to approximate these equations by expanding each in a three dimensional Taylor series. Keeping only the linear terms, we find—at least approximately—that

\[
0 = f_1 + \frac{\partial f_1}{\partial x_1} \delta x_1 + \frac{\partial f_1}{\partial x_2} \delta x_2 + \frac{\partial f_1}{\partial x_3} \delta x_3 \\
0 = f_2 + \frac{\partial f_2}{\partial x_1} \delta x_1 + \frac{\partial f_2}{\partial x_2} \delta x_2 + \frac{\partial f_2}{\partial x_3} \delta x_3 \\
0 = f_3 + \frac{\partial f_3}{\partial x_1} \delta x_1 + \frac{\partial f_3}{\partial x_2} \delta x_2 + \frac{\partial f_3}{\partial x_3} \delta x_3
\]  

(14.52)

Here, \( f_1, f_2, f_3 \), and all the derivatives are evaluated at \( x_1^{(0)}, x_2^{(0)}, x_3^{(0)} \) and are known. In effect, we have converted the problem of finding \( \delta x_1, \delta x_2 \), and \( \delta x_3 \) into one of solving a set of simultaneous linear equations! Once that solution is in hand, we take \( x_1^{(1)} = x_1^{(0)} + \delta x_1, \ldots \) as a refined approximation to the desired solution and repeat the process to obtain \( x_1^{(2)}, \ldots \), continuing until some chosen convergence criterion is satisfied.

Each of the software packages described in this book makes available a spectrum of commands to solve simultaneous nonlinear equations. We here merely enumerate those commands and indicate the general strategy each implements. We leave the descriptive details to the vendor-supplied documentation, and we leave illustrative applications to the exercises at the end of the chapter.

**NUMERICAL RECIPES in C:** For numerical solution of systems of nonlinear equations, the primary C subroutines in the *Numerical Recipes* library are `mnewt.c`, which implements a simple Newton’s method; `newt.c`, which adopts an embellished Newton’s method with improved convergence; and `broydn.c`, which implements a multidimensional secant method described for one variable in Section 14.6.3.

### 14.17 Exercises

#### 14.17.1 ... using Symbolic Methods

14.1. Derive both forms of the quadratic formula as given in Eq. (14.29) for the roots of the polynomial \( ax^2 + bx + c \). *Hint:* Start by completing the square, i.e., by adding and subtracting the right amount so that the polynomial can be expressed in the form \( a(x - \alpha)^2 + \beta \).

14.2. Find the value of \( x \) at which \( f(x) = ax^2 + bx + c \) has an extremum and determine a criterion involving the coefficients (or some of them) for deciding whether the extremum is a maximum or a minimum. Assume that \( a, b, \) and \( c \) are real.

14.3. Find the points at which the function \( f(x) = ax^3 + bx^2 + cx + d \) has (local) extrema and find a criterion involving the coefficients (or some of them) that will assure that the function has three real roots. Assume that \( a, b, c, \) and \( d \) are real.
14.4. In some quantum calculations, the need to solve the equation $x(x + 1) = l(l + 1)$ for $x$ arises. Find those roots, noting particularly that, since the equation is quadratic, there are two roots. The obvious root $x = l$ is not the only one.

14.5. Each of the three blades of a lawn mower has radius $a$. As shown in Fig. 14.9, the center blade is invariably mounted somewhat in front of the two outside ones so that the areas cut by each blade can overlap without risking collision of the blades with one another. What must be the minimum offset $x$ of the center of the middle blade from the line joining the centers of the two outer blades so that their cutting paths will overlap by an amount $y$ without collision of the blades?

14.6. For the projectile discussed in Section 14.1.4, generate a family of graphs showing $\tau(\theta)$ as a function of $\theta$ for selected fixed values of $\alpha$.

14.7. In Section 14.1.4, we deduced that the range of a projectile of mass $m$ launched with speed $v_0$ at an angle $\theta$ up from the horizontal in a medium characterized by a (linear) viscous damping coefficient $b$ could be found by (1) finding the non-zero root of the equation

$$f(\tau) = \tau - (1 + \alpha \sin \theta)(1 - e^{-\tau}) = 0$$

where $\alpha = bv_0/(mg)$, and then (2) evaluating the range from the expression

$$R(\theta, \alpha) = \frac{\cos \theta(1 - e^{-\tau})}{\alpha} = \frac{\tau \cos \theta}{\alpha(1 + \alpha \sin \theta)}$$

We could view the first of these equations as defining the function $\tau(\theta)$ implicitly. In principle, for a given $\alpha$, we could imagine solving the first equation explicitly for $\tau$ as a function of $\theta$ and substituting that solution into the second equation to find an expression for the range—again for a given $\alpha$—explicitly as a function of $\theta$ alone. If we had that expression in hand, we would find the maximum range by solving the equation $dR(\theta)/d\theta = 0$ for $\theta$ and then evaluate the range at that specific value of $\theta$. In the absence of that expression, we can still differentiate $R(\theta)$ implicitly, recognizing the (hidden) dependence of $\tau$ on $\theta$, and we can differentiate $f(\tau)$. The resulting equations together might then be combinable in a way that would lead to a determination of the maximum range more directly than the route described in the text. Pursue this idea, using a symbolic manipulating program as much as possible to simplify the calculation. The ultimate objective would be to deduce and test a procedure that leads to a numerical value for the maximum range of this projectile when $\alpha$ is given. Note: This exercise is almost certainly difficult and potentially frustrating. The author has no idea whether success awaits the persistent in this endeavor.

14.8. Find the natural frequencies for the three modes of oscillation characterizing the system that results when the system shown in Fig. 14.2 is extended to contain three objects coupled in a line. Take the four springs all to have the same spring constant but allow for the possibility that the middle object may have a mass $m'$ different from the mass $m$ of the two outside objects. In particular, measure frequencies in units of $\sqrt{k/m}$ and seek a graph showing the frequency of each of the modes as a function of $\beta = m'/m$. Hint: To help you get started and to facilitate focusing on the solution of the ODEs rather than on deriving them, note that, for three masses, the equations of motion will
be
\[ m\frac{d^2x_1}{dt^2} = -kx_1 + k(x_2 - x_1) \]
\[ m\frac{d^2x_2}{dt^2} = -k(x_2 - x_1) + k(x_3 - x_2) \]
\[ m\frac{d^2x_3}{dt^2} = -k(x_3 - x_2) - kx_3 \]

14.9. In dimensionless units, the energy shifts \( E \) that occur in the states of hydrogen for \( n = 3 \) when an external constant electric field is turned on are given by the roots of the ninth-order polynomial
\[ f(E) = E^9 - \frac{243}{2} E^7 + \frac{59049}{16} E^5 - \frac{531441}{16} E^3 \]
Find the distinct roots of this polynomial and the multiplicity of each root.

14.10. If, when divided by a single line into a square and a rectangle, the resulting smaller rectangle has the same aspect ratio as the original rectangle, the rectangle is called a golden rectangle and the ratio of the longer to the shorter side is called the golden ratio. Rectangles having that ratio are considered to be particularly aesthetic and can be found in many places, including ancient Greek architecture and, intriguingly, in standard, present-day plastic credit cards. Express the above defining criterion to find a quadratic equation for the golden ratio and then solve the equation to find that ratio.

14.17.2 ... using Numerical Methods

14.11. Use numerical methods to solve Eq. (14.6) for the equilibrium points in the potential energy in Eq. (14.3).

14.12. One way to find the square root of a (positive) number \( a \) is to find the root of the function \( f(x) = x^2 - a \). (a) Apply Newton’s method symbolically to show that \( x_{n+1} = (x_n + a/x_n)/2 \). (b) Using a pocket calculator and starting with the guess \( x_0 = 2 \), work out the first few iterates by hand and note how quickly this algorithm converges to \( \sqrt{2} \approx 1.41421 \). (This algorithm is the algorithm that most pocket calculators invoke when the square root key is pressed.) (c) Using whatever computational tool appeals to you, write a program that asks for the value of \( a \), an initial guess for \( \sqrt{a} \), and a tolerance and then implements Newton’s method to find \( \sqrt{a} \), printing out each iterate along the way and stopping automatically when successive iterates differ by less than the specified tolerance.

14.13. For the first six Legendre polynomials \( L_n(x) \), find all roots lying in the interval \(-1 \leq x \leq 1 \). Those polynomials are
\[
L_0(x) = 1 \quad L_4(x) = \frac{1}{8}(35x^4 - 30x^2 + 3) \\
L_1(x) = x \quad L_5(x) = \frac{1}{8}(63x^5 - 70x^3 + 15x) \\
L_2(x) = \frac{1}{4}(3x^2 - 1) \quad L_6(x) = \frac{1}{16}(231x^6 - 315x^4 + 105x^2 - 5) 
\]

14.14. The natural frequencies for the transverse vibrations of a bar of uniform cross section that has length \( L \) and is free at both ends are given by
\[ \omega_n = \frac{4K}{L^2} \sqrt{\frac{E}{\rho}} \alpha_n^2 \]
where \( K \) is the radius of gyration of the cross section of the bar, \( E \) is Young’s modulus for the material of the bar, \( \rho \) is the density (mass/unit volume) of the material of the bar, and \( \alpha_n \) is a solution to the equation
\[ \tan \alpha = \pm \tanh \alpha \]
14.17. **EXERCISES**


14.15. If the bar of the previous exercise is clamped at one end and free at the other, then the natural frequencies are given by the same expression except that $\alpha_n$ is instead a solution to the equation

$$\cot \alpha = \pm \tanh \alpha$$


14.16. The natural frequencies of the air in a spherical cavity are determined from the roots of the function $14.16$. Explore the way the energy levels of the well described in Section 14.1.5 change as the parameter $c$, which is determined by the depth and the width of the well, increases. At base, changing $c$ changes the slopes of the straight lines in Fig. 14.5. As $c$ increases and the well becomes deeper, the lines become more and more nearly horizontal and the number of energy levels increases. Seek ultimately to generate a graph that shows the energy of each allowed level on the vertical axis as a function of the parameter $c$ along the horizontal axis.

14.17. The intensity $I(x)$ in the diffraction pattern produced by a single slit is given by

$$I(x) = \frac{\sin^2 x}{x^2}$$

where $I_0$ is the intensity in the center and $x$ is related to the position of the observation point away from the central maximum. The *zeros* in this pattern are easy to locate (they occur at $x = n\pi$, $n = 0, \pm 1, \pm 2, \ldots$). Careful location of the *maxima*, however, is more complicated. They don’t occur where $\sin^2 x = 1$ because of the influence of the denominator that steadily increases as $x$ increases. Locate the positions of the first half dozen maxima in this pattern, which—basically—is a request to find the roots of the derivative of the function (though note that not all roots correspond to maxima). Use at least three different methods and at least two different computational tools, and compare the results. Do your results confirm that the roots approach odd multiples of $\frac{1}{2}\pi$ as they become large? *Optional:* You might also find it interesting to approximate the function with a power series expansion for $\sin x$, keeping quite a few terms but converting the root finding problem into that of finding the roots of a polynomial. Then, use methods for finding roots of polynomials and see if you can come to understand how accuracy depends on how many terms you keep and which root you seek.

14.19. Using at least three different methods and at least two different computational tools, find the first half dozen roots of the zeroth-order Bessel function $J_0(x)$. Note that these roots are related to the radii of circular nodes in some of the vibrations of a circular membrane. The values of these roots tabulated in Abramowitz and Stegun are $2.404825577$, $5.5200781103$, $8.6537279129$, $11.7915344391$, $14.9309177086$, $18.0710639679$. *Hint:* Most computational tools have built-in capabilities for evaluating the Bessel functions. Consult the appropriate vendor manuals.

14.20. Suppose a particle moves in one dimension under the influence of the potential energy

$$V(x) = -\frac{V_0 a^2 (a^2 + x^2)}{8a^4 + x^4}$$

then

$$\frac{V(x)}{V_0} = -\frac{(1 + \frac{x^2}{a^4})}{8 + \frac{1}{a^4}}.$$

---

where \( \bar{x} = x/a \). Using at least three different methods and at least two different computational tools, find the coordinates \( \bar{x} \) of all turning points when the total energy \( E \) of the particle is \( E = -0.2V_0 \) and also when the total energy is \( E = -0.1V_0 \). Optional: Obtain graphs of the position of each turning point as a function of particle energy over the allowed range of energies for bound states.

14.21. Generate a graph showing the turning points of the potential energy given by Eq. (14.3) as a function of the energy of the particle.

14.22. Suppose a straight railroad track of length 1 mile (5280 ft) is held absolutely immovable at its two ends. On a hot summer day, the track expands in length by 1 ft. If the track bows upward from the earth in a circular arc, how high above the earth will the track rise at its midpoint? \textit{Hint}: The geometry of this exercise is shown in the accompanying figure, where \( l \) is the original length of the track, \( d \) is the rise at the center, and \( a \) is the length, \( R \) the radius, and \( \theta \) half the subtended angle of the circular arc. Thus, \( R\theta = a/2, \sin \theta = l/(2R) \), and \( R - d = R \cos \theta \). The task is to find \( \theta \) and \( R \) from the first two of these equations and then use the third to find \( d \). The only quantities known \textit{a priori} are \( l \) and \( a \).

14.23. A particle of mass \( m \) moves in a potential energy given as a function of position by \( V(x) = V_0 \cosh(x/a) \). Because this potential energy is an even function of \( x \), the upper and lower turning points have the same absolute value but opposite sign. Find the upper turning point as a function of energy and generate a graph showing that turning point as a function of energy. \textit{Hint}: Measure position in units of \( a \) and energy in units of \( V_0 \).

14.24. A particular problem—see Problem 3-19 in the fourth edition of \textit{Fluid Flow} by Rolf H. Sabersky, Allan J. Acosta, Edward G. Hauptmann, and E. M. Gates (Prentice-Hall, Upper Saddle River, NJ, 1999)—in fluid flow leads to the need to find the roots of the fourth-order polynomial \( 12x^4 - 12x^3 + 4x - 1 \). Use graphical methods to find bounds on the roots and at least three different computational approaches to find all real roots of this polynomial.

14.26. Write and test a C program paralleling \texttt{bisect.c} but using Newton’s method to find the roots of \( f(x) \). Your program, which you might call \texttt{newton.c}, should

- use the functions \texttt{FUNC} and \texttt{FUNCD} to return \( f(x) \) and \( df(x)/dx \), respectively.
- request a tolerance, an initial guess, a maximum number of iterations, and a flag—0 or 1—to be entered when run.
- find the root, terminating iteration either when successive iterates differ by less than the specified tolerance or when the specified maximum number of iterations is exceeded.
- print the final iterate only (flag = 0) or all iterates along the way (flag = 1).
- print a warning if iteration is terminated because the maximum number of iterates was exceeded.
- print the root and the value of the function at that root.

14.27. The image of a distant object produced on a viewing screen by a small aperture is actually a diffraction pattern. When the aperture is a circle of \textit{diameter} \( d \) and light from the object strikes the screen containing the aperture at normal incidence, the intensity in the diffraction pattern at angle \( \theta \) from the normal is given by

\[
I(\theta) = \frac{2J_1(\pi d \sin \theta / \lambda)}{\pi d \sin \theta / \lambda}^2
\]

where \( I_0 \) is the intensity at the center of the pattern, \( \lambda \) is the wavelength of the light illuminating the aperture, and \( J_1(x) \) is the first order Bessel function. Using the Bessel function routine that
14.17. EXERCISES

14.28. The Lennard-Jones potential energy $V_{LJ}$, which is given in terms of the coordinate $r$ by the expression

$$\frac{V_{LJ}}{\epsilon} = 4 \left( \frac{\sigma^{12}}{r^{12}} - \frac{\sigma^6}{r^6} \right) \quad \text{or} \quad \frac{V_{LJ}}{\epsilon} = 4 \left[ \frac{1}{r^{12}} - \frac{1}{r^6} \right]$$

where $\epsilon$ and $\sigma$ are constants and $\bar{r} = r/\sigma$, plays a prominent role in some theories of chemical bonding. Obtain a graph of this potential energy and then obtain graphs of the lower turning point and the upper turning point as functions of the energy of the system over the range of energies for which the particle experiencing the potential energy is bound in the associated potential well.

14.17.3 ... using Numerical Recipes

Note: Numerical recipes can, of course, also be used for any of the exercises in Section 14.17.2.

14.30. Study the demonstration programs xrtbis.c and xrtnewt.c, each of which is written so as to be able to find all roots of a given function in a specified range. Then recast each of these programs to produce a program that will find all of the positive roots of the equation for each sign in Eq. (14.25). Write your program so that the value of $c$ and the boundaries of the region in which roots are to be sought are entered from the terminal at the time the program is run. Finally, test your program with $c = 25$, comparing your results with those obtained in the text, and then examine other values of $c$, seeking ultimately to obtain a graph showing each root as a function of $c$.

14.32. Read the Numerical Recipes books about the recipe rtsafe.c and the example program xrtsafe.c. Then recast the program sinrtbis.c (or sinrtnewt.c) to use rtsafe.c and test your program with a couple of functions of your choice.

14.17.4 Finding More than One Unknown

14.33. In a global positioning system, the raw data from which the position is determined consists of distances from various reference points together with knowledge of the location of those reference points. In two dimensions, for example, we might try to locate our position $(x, y)$ in a plane from knowledge that we are a distance $r_1$ from the point $(x_1, y_1)$ and a distance $r_2$ from the point $(x_2, y_2)$. Not all values we might assign are physically meaningful. For example, there is no point that is simultaneously 20 miles from point 1 and 30 miles from point 2 if points 1 and 2 are, in fact, 100 miles apart. Depending on the circumstances, there may be no points, one point, or—most often—two. Develop an algorithm for finding your location in two dimensions when you know your distance from each of two reference points whose coordinates are known, implement your algorithm in a program using whatever computational tool seems appropriate, solve two or three test problems that you invent, and—in particular—try to describe and defend the conditions under which two, one, or no solutions exist. Optional: Extend your entire consideration into three dimensions, which will require knowledge of distance from each of (at least) three known locations. Hint: Quite a bit of information is available on the website www.trimble.com/gps. There is also an article in the January, 1994, issue of Physics Today (“Where I Stand” by Daniel Kleppner, page 9).

14.34. Given the three points $(x_i, y_i)$, $i = 1, 2, 3$, (a) find symbolic expressions for the coefficients $a$, $b$ and $c$ of the parabola $y = ax^2 + bx + c$ that passes through these three points and then (b) find a symbolic expression for the value of $x$ at which the extremum point of the parabola occurs. Finally, (c) determine numerically the angle at which the maximum range of a projectile occurs if the ranges at $\theta = 30^\circ$, $40^\circ$, and $41^\circ$ are 0.7251744, 0.7250484, and 0.7258887, respectively.
14.35. Kirchhoff’s laws in DC circuit theory contend that the net current flowing into any node must be zero and that the net voltage drop around any closed path in the circuit must be zero. Remembering that the voltage drop $\Delta V$ across a resistor $r$ carrying current $i$ is given by $\Delta V = ir$ and using the symbols defined in Fig. 14.10, apply these laws to each of the circuits in the figure to generate a set of simultaneous, inhomogeneous linear equations for the unknown currents. Then, using symbolic methods solve each case for the unknown currents. Finally, determine the effective resistance defined by $R_{\text{eff}} = V/I$ for each circuit. Assume that all batteries and resistors have known values and that quantities represented in the figures with the same symbol have the same value. Warning: For even simple circuits, Kirchhoff’s laws provide more equations than unknowns. Correctly written, these equations are guaranteed to be consistent. The subset to be solved, however, must be carefully chosen to make sure its members are linearly independent of one another.

14.36. The file $\$HEAD/data/freefall.dat$ contains 31 lines, the $i$-th of which contains one numerical value—the value of the position $x_i$ in cm of a particle at time $t_i = (i-1)/60$ s. You have reason to believe that the set of data $(t_i, x_i)$ with $i = 1, 2, 3, \ldots, n$ is described by the parabolic relationship

$$x = at^2 + bt + c$$

The method of least squares identifies the optimum values of the coefficients $a$, $b$, and $c$ as the particular values that minimize the residual

$$R(a, b, c) = \sum_{i=1}^{n} \left( x_i - (at_i^2 + bt_i + c) \right)^2$$

Using whatever language you choose, write a program that reads the positions from the file into a vector, generates a second vector containing the corresponding times and then enters a loop in which it asks for entry of trial values of $a$, $b$, and $c$, calculates and displays $R(a, b, c)$, and returns to ask for a new set of trial values. Using this program, conduct a manual search for the values of $a$, $b$, and $c$. 

Figure 14.10: Circuits for Exercise 14.35.
14.17. EXERCISES

14.17. EXERCISES

14.37. The file $\$HEAD/data/freefall.dat$ contains 31 lines, the i-th of which contains one numerical value—the value of the position $x_i$ in cm of a particle at time $t_i = (i-1)/60$ s. You have reason to believe that the set of data $(t_i, x_i)$ with $i = 1, 2, 3, \ldots, n$ is described by the parabolic relationship

$$x = at^2 + bt + c$$

The method of least squares identifies the optimum values of the coefficients $a$, $b$, and $c$ as the particular values that minimize the residual

$$R(a, b, c) = \sum_{i=1}^{n} \left( x_i - (at_i^2 + bt_i + c) \right)^2$$

i.e., as the particular values satisfying the three equations

$$\frac{\partial R}{\partial a} = 0 ; \frac{\partial R}{\partial b} = 0 ; \frac{\partial R}{\partial c} = 0$$

These equations will turn out to be linear in $a$, $b$, and $c$, with coefficients and inhomogenieties determined by sums of various products of the measured independent and dependent variables. Derive the three equations symbolically but then find the numerical values of the coefficients and inhomogenities for the data in the file $\$HEAD/data/freefall.dat$ and use at least two different numerical approaches to find the solution for $a$, $b$, and $c$. Finally, generate a graph in which each measured point is represented by a simple symbol and the least squares parabola is shown by a solid line so you can judge the adequacy of your fit.

14.38. A particle of mass $m$ moves along the $x$ axis under the action of a time-dependent force $F(t)$. We observe that $x(0) = 0$ and $x(t_f) = a$. The detailed motion therefore is described by the solution to the boundary value problem

$$m \frac{d^2x}{dt^2} = F(t) , \quad x(0) = 0 , \quad x(t_f) = a$$

To predict the detailed motion numerically, we might divide the interval $0 \leq t \leq t_f$ into $n$ segments of size $\Delta t = t_f/n$, let $x_i = i \Delta t$ with $i = 0, 1, 2, \ldots, n$, evaluate the differential equation at $t = t_i$, and introduce a finite difference approximation for the derivative to conclude that

$$x_{i-1} - 2x_i + x_{i+1} = \frac{F(t_i)}{m} \Delta t^2$$

which is valid for $i = 1, 2, \ldots, n-1$. For the two end points, we remember the boundary values and require that

$$x_0 = 0 , \quad x_n = a$$

In total, we have $n + 1$ equations determining the $n + 1$ unknowns $x_0, x_1, x_2, \ldots, x_n$. Cast these equations in the matrix form

$$\begin{pmatrix}
? & ? & ? & \ldots & ? \\
? & ? & ? & \ldots & ? \\
? & ? & ? & \ldots & ? \\
\vdots & \vdots & \vdots & \ldots & \vdots \\
? & ? & ? & \ldots & ? \\
\end{pmatrix} \begin{pmatrix}
x_0 \\
x_1 \\
x_2 \\
\vdots \\
x_n \\
\end{pmatrix} = \begin{pmatrix}
? \\
? \\
? \\
\vdots \\
? \\
\end{pmatrix}$$

and note that the coefficient matrix is tridiagonal (and symmetric). Then, using at least two different computational tools, design and describe a general approach and implement that approach to determine the detailed motion for the cases

(a) $t_f = 1$ s, $a = 10$ m, $F(t)/m = 8$ N/kg (constant force), and $n = 10$;
(b) same as (a) but with $n = 100$;
14.39. The electrostatic potential on the boundary of the square region in a plane shown in Fig. 14.11 is maintained at the value zero along its bottom and right edges but increases linearly from zero along the left edge, then decreases linearly back to zero along the top edge. The potential in the interior region satisfies Laplace’s equation. One approach to finding an approximate solution involves imposing a regular grid on the region and then requiring that the value of the potential at each interior point be the average of the values at its four nearest neighbors, e.g.,

\[ V_{12} = \frac{1}{4} \left( V_{11} + V_7 + V_{13} + V_{17} \right) \]

(a) Write out the twenty-five equations implied by this requirement, (b) cast them in the matrix form

\[
\begin{pmatrix}
? & ? & ? & \ldots & ? \\
? & ? & ? & \ldots & ? \\
? & ? & ? & \ldots & ? \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
? & ? & ? & \ldots & ?
\end{pmatrix}
\begin{pmatrix}
V_1 \\
V_2 \\
V_3 \\
\vdots \\
V_{25}
\end{pmatrix}
= 
\begin{pmatrix}
? \\
? \\
? \\
\vdots \\
?
\end{pmatrix}
\]

figuring out the value represented by each question mark, and (c) solve the system for the unknowns. Then (d) structure the boundary values and solution in a 7 × 7 matrix and create a surface plot of that solution.

14.40. In Ruchardt’s experiment, a steel ball bounces up and down in a vertical tube that ends in a gallon (or larger) jug. Ultimately, the ball falls into the jug, but it may bounce up and down many times before doing so. The file $\text{HEAD/data/ruchardt.dat}$ contains 170 lines, the i-th of which contains one numerical value—the position \( x_i \) in cm of the steel ball at time \( t_i = 0.05(i - 1) \) s. A quick graph of the data (you should make it) suggests that the motion might be described by an exponentially decaying cosine curve on which is superimposed a linear sinking of the “equilibrium” position, i.e.,

\[ x(t) = Ae^{-bt} \cos(\omega t + \phi) + Ct + D \]

(c) \( t_f = 1 \) s, \( a = 10 \) m, \( F(t)/m = e^{-t} \) N/kg (exponentially decaying force; \( t \) in seconds), and \( n = 10; \) and

(d) same as (c) but with \( n = 100. \)
where the parameters $b$, $A$, $\omega$, $\phi$, $C$, and $D$ are to be determined. Develop a means to find optimal values of these six parameters by seeking values that minimize the quantity

$$R = \sum_{i=1}^{N} \left( x_i - Ae^{-bt_i} \cos(\omega t_i + \phi) + Ct_i + D \right)^2$$

Warning: This exercise is not for the faint-hearted. Some useful background will be found in Chapter 8 of *Data Reduction and Error Analysis for the Physical Sciences* (Second Edition) by Philip R. Bevington and D. Keith Robinson (McGraw-Hill, New York, 1992).
14.E Listing of bisect.c

/* PROGRAM bisect.c

This C program asks for the lower and upper bounds of an interval in which a single root of a function is known to exist and then invokes the method of bisection to converge on that root. Refinement of the interval continues until the tolerance TOL---set early in the main program---is reached. The function can be changed by editing its definition in the first lines of code.
*/

#include <stdio.h>  /* Load standard i.o routines */
#include <math.h>   /* Load standard math routines */

/****** DEFINE FUNCTION ******/

float func( float x )
{
    return pow(x,3)/10000.0 + pow(x,2)/200.0 - x/500.0 - 0.5;
}

void main()
{
    float tol, xa, xm, xb;   /* For tolerance, for bounds */
    float fa, fm, fb;        /* For values of function */

    tol = 0.0001;

    /****** READ BRACKET VALUES ******/

    printf( "\nLower bound = " ); scanf( "%f", &xa );
    printf( "\nUpper bound = " ); scanf( "%f", &xb );

    /****** EVALUATE FUNCTION AT EXTREMES ******/

    fa = func(xa); fb = func(xb); fm = fa;
    if (fa*fb >= 0.0) { printf( "No root in interval\n" ); return; };

    /****** HUNT FOR ROOT ******/

    while ( (xb-xa) >= tol & fm != 0.0 )
    { xm = 0.5*(xb+xa);
        fm = func(xm);
        if (fa*fm >= 0.0) xa = xm; else xb = xm;
    }

    /****** OUTPUT RESULTS ******/

    printf( "root = %15.6f\n", xm );
    printf( "xb - xa = %15.6f\n", xb-xa );
    printf( "function = %15.6f\n", func( xm ) );
}
Chapter 15

Solving Partial Differential Equations

The properties—temperature, electrostatic potential, magnetic field, membrane displacements, quantum wave functions, fluid pressure, ...—of physical systems are usually encapsulated mathematically in functions of space and time. As the properties are constrained physically by the systems themselves, the functions representing the properties are constrained mathematically by various physical laws, which typically specify one or more relationships among the partial derivatives of the dependent variable(s) with respect to two or more independent variables and hence will take the form of one or more partial differential equations (PDEs). The important PDEs of mathematical physics are second-order equations and are frequently—though certainly not always—linear. In this chapter, we limit ourselves to linear equations.

To define a problem completely (i.e., so that it has a unique solution), the associated PDEs must be supplemented with appropriate boundary conditions, which will usually fall into one of three categories, specifically,

- Dirichlet boundary conditions or boundary conditions of the first kind, which specify values that the solution must assume on the boundary of the region in which a solution is sought,
- Neumann boundary conditions or boundary conditions of the second kind, which specify values that a derivative of the solution must assume on the boundary of that region, and
- mixed boundary conditions or boundary conditions of the third kind, which specify values that some linear combination of the solution and its derivatives must assume on the boundary of that region.

When time is among the independent variables, complete specification of a problem will also entail stipulation of appropriate initial conditions, which specify values that the function and—for equations that are second order in time—its time derivative must assume throughout the spatial domain at a specific time (usually taken to be time 0). Depending on which conditions are necessary, we face a boundary-value problem (BVP) or an initial-value problem (IVP) or, sometimes, a problem involving both boundary and initial conditions.

Although closed form, symbolic solutions exist for some BVPs and IVPs, most of the time such problems can only be solved approximately by numerical methods. Most commonly, approaches to the numerical solution of PDEs involve (1) selecting a (usually large) set of points or nodes (or, in some contexts, vertices) that cover the ranges of all (or all save one) of the independent variables, (2) approximating the PDE(s) in such a way as to convert it (them) into a (usually large) set of algebraic equations (AEs) or ordinary differential equations (ODEs), and (3) solving the resulting set
of AEs or ODEs for the (approximate) solution to the original problem at each node. For example, rather than seeking the solution \( u(x, y, t) \) to the diffusion equation in two dimensions, we might introduce a grid or mesh, i.e., a set of discrete points \((x_i, y_j, t_k)\) distributed somehow over the ranges of the variables, and then convert the PDE into a set of algebraic equations for the several values \( u_{i,j,k} = u(x_i, y_j, t_k) \). Alternatively, we could discretize only the spatial variables by introducing a grid defined by the points \((x_i, y_j)\) and convert the PDE into a set of ODEs for the several functions \( u_{i,j} = u(x_i, y_j, t) \) of the continuous variable \( t \). In either case, the boundary and initial conditions constrain the values and/or derivatives of the solution at nodes on the boundary. Thus, the number of independent AEs or ODEs will turn out to be exactly equal to the number of unknown values or functions in the problem.

At least two distinctly different approaches to the conversion of a PDE into a set of AEs or ODEs are in common use. Finite difference methods (FDMs) are quick, (comparatively) easy to motivate, and fast to code in computer languages, but they are borderline impossible to apply unless (1) the nodes are uniformly spaced and (2) the boundaries of the region in which a solution is sought coincide with the coordinate lines in one of the standard coordinate systems (Cartesian, polar, cylindrical, spherical, ...). An algebraic expression, which references two or more adjacent nodes, is used to approximate each derivative in the differential equation at each of these nodes. Ultimately, the partial differential equation(s) is (are) replaced by a system of AEs, (or, in some cases, by a system of ODEs), which is then solved for the dependent variable at each node.

In finite element methods (FEMs), however, the nodes forming the grid may be non-uniformly spaced. Collections of these nodes form geometric shapes—lines in one dimension; triangles or quadrilaterals or ... in two dimensions; tetrahedrons or hexahedrons (sometimes called bricks) or ... in three dimensions—that divide the region of interest into subregions, i.e., into the (finite) elements that give the method its name. Interpolation or shape functions are defined to facilitate approximating the dependent variable at points within each element from presumed known values of the dependent variable at the nodes or vertices associated with that element. The differential equation is then replaced with an equivalent integral statement, which is in turn converted into a system of algebraic equations by substituting the shape functions into this integral form, integrating, assembling the results from all elements, and imposing the constraints dictated by the boundary and initial conditions. Thus, as with the finite difference approach, the original partial differential equation(s) is (are) replaced by a system of algebraic equations, which is then solved for the dependent variable at each node.

Finite element methods have significant advantages over finite difference methods. For example, since finite element methods allow a nonuniform mesh, portions of the region in which a solution is sought can be treated with a fine mesh while, at the same time, other portions of that region can be represented with a coarse mesh, so the computational effort can be focused where it is most needed. Furthermore, especially if the boundaries are irregular, the boundary conditions are more easily incorporated with finite element methods than with finite difference methods. While we gain considerably in generality by invoking a method of this second type, we pay a heavy price: finite element methods are difficult to describe and even more difficult to code.

In Section 15.1, we deduce and provide physical contexts for a number of important PDEs. Then, in subsequent sections, we introduce both finite difference and finite element methods, illustrating each with explicit solutions to representative problems. For the sake of the quickest exposition of the essential ideas, we start by addressing problems with one independent variable before moving on to problems having two or more independent variables and requiring coordinate systems other than Cartesian. Because the methods are elaborate and their coding in computer languages is involved and lengthy, this chapter will undoubtedly seem more mathematical and abstract than physical. However obscured it may appear to be by the detailed discussion of method and implementation, the desire to address physical situations does assuredly lie underneath the entire exposition.
15.1 Sample Problems

We begin by laying out several of the most common PDEs of mathematical physics, placing each in at least some of the physical contexts in which it appears. As we shall discover in Section 15.1.5, the most common equations fall into one of three categories, represented respectively by the (classical) wave equation, the diffusion equation, and the Laplace equation.

15.1.1 Motion of a String: The Wave Equation

Consider a string that extends along the (horizontal) $x$ axis when it is in its equilibrium position, and let a point of the string when the string is in that position be located at $x$. Suppose that the string is moving in a plane (not the most complicated possible motion) such that, at a general time $t$, the point nominally located at $x$ is displaced transversely (i.e., perpendicular to the equilibrium orientation of the string)\(^1\) by an amount $u(x,t)$ and longitudinally (i.e., parallel to that equilibrium orientation) by an amount $v(x,t)$. The geometry is shown in Fig. 15.1. The string is, of course, under tension $\tau(x,t)$, which in general will vary from point to point and from time to time but, for a perfectly flexible string (which we explicitly assume), will always be directed tangent to the string. A force diagram for an isolated element—the element located between $x$ and $x+\Delta x$ when the string is in equilibrium—of the string might look like Fig. 15.2. For the transverse motion of the string, Newton’s second law $\mathbf{F} = ma$ then requires that

$$\rho(x) \Delta x \frac{\partial^2 u}{\partial t^2} = \tau(x+\Delta x, t) \sin \theta(x+\Delta x, t) - \tau(x, t) \sin \theta(x, t) - \rho(x) \Delta x g$$  \hspace{1cm} (15.1)

or, if we divide by $\Delta x$ and let $\Delta x$ approach zero, that

$$\rho(x) \frac{\partial^2 u}{\partial t^2} = \frac{\partial}{\partial x} \left( \tau(x, t) \sin \theta(x, t) \right) - \rho(x) g$$  \hspace{1cm} (15.2)

\(^1\)We take this transverse direction to be vertical, and include gravity among the forces acting on the string.
Figure 15.3: Geometry for determining $\theta(x, t)$ from $u(x, t)$ and $v(x, t)$.

Here, $\rho(x) \Delta x$ is the mass of the element, and $\rho(x) g \Delta x$—the gravitational force on the element when the string is near the surface of the earth—illustrates a possible external force on the string. Similarly, for the longitudinal motion of the string, Newton’s second law requires that

$$\rho(x) \Delta x \frac{\partial^2 v}{\partial t^2} = \tau(x + \Delta x, t) \cos \theta(x + \Delta x, t) - \tau(x, t) \cos \theta(x, t)$$

(15.3)

or that

$$\rho(x) \frac{\partial^2 v}{\partial t^2} = \frac{\partial}{\partial x} \left( \tau(x, t) \cos \theta(x, t) \right)$$

(15.4)

for the horizontal motion.

This set of equations still contains too many unknowns. We must eliminate $\theta(x, t)$, because we really want to find only $u(x, t)$ and $v(x, t)$. Figure 15.3 supports the conclusion that

$$\sin \theta = \frac{\Delta u}{\Delta s} = \frac{\Delta u}{\sqrt{\Delta u^2 + (\Delta x + \Delta v)^2}} = \frac{\partial u}{\partial x} \left( \frac{\partial u}{\partial x} \right)^2 + \left( 1 + \frac{\partial v}{\partial x} \right)^2 \right]^{1/2}$$

(15.5)

and that

$$\cos \theta = \frac{\Delta x + \Delta v}{\Delta s} = \frac{1 + \frac{\partial v}{\partial x}}{\left[ \left( \frac{\partial u}{\partial x} \right)^2 + \left( 1 + \frac{\partial v}{\partial x} \right)^2 \right]^{1/2}}$$

(15.6)

Thus, the equations describing the motion of the string are

$$\rho(x) \frac{\partial^2 u}{\partial t^2} = \frac{\partial }{\partial x} \left[ \tau(x, t) \frac{\partial u}{\partial x} \left( \frac{\partial u}{\partial x} \right)^2 + \left( 1 + \frac{\partial v}{\partial x} \right)^2 \right]^{1/2} - \rho g$$

(15.7)

$$\rho(x) \frac{\partial^2 v}{\partial t^2} = \frac{\partial }{\partial x} \left[ \tau(x, t) \frac{\left( 1 + \frac{\partial v}{\partial x} \right)^2}{\left[ \left( \frac{\partial u}{\partial x} \right)^2 + \left( 1 + \frac{\partial v}{\partial x} \right)^2 \right]^{1/2}} \right]$$

(15.8)
The system is, of course, not complete as yet. We would also need to know not only appropriate initial conditions, i.e., initial values for the four functions

\[ u(x,0), \quad v(x,0), \quad \frac{\partial u}{\partial x}(x,0), \quad \frac{\partial v}{\partial x}(x,0) \] (15.9)

but also appropriate boundary conditions on the string at its two ends and a connection between \( \tau \) on the one hand and \( u \) and \( v \) on the other. This system is decidedly non-linear and very difficult to solve.

Approximations are almost always necessary to turn the problem we would really like to solve into one that we can solve. If, for example, (1) the amplitude of the motion is small so that

\[ \Delta u \ll \Delta x \implies \frac{\partial u}{\partial x} \ll 1 \] (15.10)

(2) \( \tau \) is sufficiently large that, in small amplitude motion, \( \tau \) remains essentially constant, and (3) the motion is transverse so that \( v = 0 \) everywhere and always, then the equation for \( v \) is automatically satisfied and the equation for \( u \) decouples from that for \( v \) and becomes the (inhomogeneous) wave equation,

\[ \rho \frac{\partial^2 u}{\partial t^2} = \tau \frac{\partial^2 u}{\partial x^2} - \rho g \] (15.11)

which is correct under the given circumstances even if \( \rho \) varies with \( x \).

We have derived this result for motion of a string in one dimension. Two special cases are worth noting:

- If the situation is static so there is no time dependence (\( \partial u / \partial t = 0 \)), then the shape of a string hanging under its own weight is given by the solution to the equation

\[ \tau \frac{d^2 u}{dx^2} = \rho g \] (15.12)

which reduces in this case to an ODE.

- If there is no outside force and \( \rho \) is constant, then Eq. (15.11) becomes

\[ \rho \frac{\partial^2 u}{\partial t^2} = \tau \frac{\partial^2 u}{\partial x^2} \implies \frac{\partial^2 u}{\partial t^2} = \frac{\tau}{\rho} \frac{\partial^2 u}{\partial x^2} \implies \frac{1}{c^2} \frac{\partial^2 u}{\partial t^2} = \frac{\partial^2 u}{\partial x^2} \] (15.13)

Note that the constant \( c = \sqrt{\tau/\rho} \) has the units of velocity.\(^2\) It turns out—though we don’t know this as yet—that \( c \) is the speed of propagation of the wave conveyed by the solution to this equation.

In both cases, of course, appropriate boundary and initial conditions must be specified before the equation has a unique solution.

While its deduction is more complicated, Eq. (15.13) has a natural extension to two and three dimensions, in which case we would consider, for example, the displacement of a two-dimensional membrane or the pressure in a gas in a three-dimensional enclosure. The function we seek would then be \( u = u(x,y,z,t) = u(r,t) \) and the second derivative with respect to one spatial dimension becomes the Laplacian. The equation in that case is

\[ \frac{1}{c^2} \frac{\partial^2 u}{\partial t^2} = \nabla^2 u \quad \text{or} \quad \left( \nabla^2 - \frac{1}{c^2} \frac{\partial^2}{\partial t^2} \right) = 0 \] (15.14)

\(^2\)Since \( [\tau] = \text{N} = \text{kg} \cdot \text{m/s}^2 \) and \( [\rho] = \text{kg}/\text{m} \), \( [\tau/\rho] = (\text{kg} \cdot \text{m}/\text{s}^2)/(\text{kg}/\text{m}) = \text{m}^2/\text{s}^2 \) (Here, the symbol \([\ldots]\) stands for the units of \( \ldots \)).
Here

\[ \nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2} \quad (15.15) \]

\[ = \frac{1}{r} \frac{\partial}{\partial r} \left( r \frac{\partial}{\partial r} \right) + \frac{1}{r^2} \frac{\partial^2}{\partial \phi^2} + \frac{\partial^2}{\partial z^2} \quad (15.16) \]

\[ = \frac{1}{r^2} \frac{\partial}{\partial r} \left( r^2 \frac{\partial}{\partial r} \right) + \frac{1}{r^2 \sin \theta} \frac{\partial}{\partial \theta} \left( \sin \theta \frac{\partial}{\partial \theta} \right) + \frac{1}{r^2 \sin^2 \theta} \frac{\partial^2}{\partial \phi^2} \quad (15.17) \]

where \((x, y, z)\), \((r, \phi, z)\), and \((r, \theta, \phi)\) are the physicists’ standard Cartesian, cylindrical, and spherical coordinates. We thus arrive at the (classical) wave equation, one of the three prototype equations in mathematical physics.

### 15.1.2 Heat Flow: The Diffusion Equation

To motivate a second of the important equations, we seek the temperature \(u(x, t)\) in the one-dimensional rod shown in Fig. 15.4. Let the rod be insulated along its sides and characterized by a thermal conductivity \(K\), which gives the rate of heat flow per unit area in the direction of increasing \(x\) by the expression

\[ \text{rate of heat flow in rod towards positive } x \text{ through cross section at right angles to rod} = -KA \frac{\partial u}{\partial x} \quad (15.18) \]

where \(A\) is the area of a cross section of the rod. [In effect, Eq. (15.18) defines \(K\).] Here, the derivative \(\partial u/\partial x\) is the gradient of the temperature, and the minus sign appears because heat flows in the positive direction when the gradient is negative and in the negative direction when the gradient is positive, i.e., heat flows from regions of higher temperature to regions of lower temperature; the explicit minus sign then assures that \(K\) will be a positive quantity. With this definition of a material property, which may depend on position in the rod, we conclude that, as heat flows in the rod,

\[ \text{heat flow in time } \Delta t \text{ into shaded element across surface at } x = -KA \left. \frac{\partial u}{\partial x} \right|_x \Delta t \quad (15.19) \]

and

\[ \text{heat flow in time } \Delta t \text{ into shaded element across surface at } x + \Delta x = +KA \left. \frac{\partial u}{\partial x} \right|_{x+\Delta x} \Delta t \quad (15.20) \]

so that, in the end,

\[ \text{net heat conducted into shaded element in time } \Delta t = \left[ \left( KA \frac{\partial u}{\partial x} \right)_{x+\Delta x} \right] - \left( KA \frac{\partial u}{\partial x} \right)_{x} \Delta t \quad (15.21) \]
This energy, of course, will affect the temperature of the element. Indeed, if we introduce two other properties of the material, its heat capacity per unit mass \( c \), and its density \( \rho \), we then conclude that

heat necessary to increase temperature of shaded element by amount \( \Delta u \) = \((A \Delta x \rho c)\Delta u\) \hspace{1cm} (15.22)

where \( A \Delta x \) is the volume of the element, and (hence) \( \rho A \Delta x \) is its mass. Since any heat transported into the element must affect its temperature as per this equation, these two evaluations of the heat flux must be equal, and we conclude that

\[
\rho c \frac{\partial u}{\partial t} = \frac{\partial}{\partial x} \left( K \frac{\partial u}{\partial x} \right) \hspace{1cm} (15.23)
\]

or, if \( K \) happens to be constant, that

\[
\frac{\rho c}{K} \frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} \implies \frac{\partial^2 u}{\partial x^2} = \frac{1}{\alpha^2} \frac{\partial u}{\partial t} \hspace{1cm} (15.24)
\]

where \( \alpha^2 = K/\rho c \). The statement of the problem is, of course, not complete until appropriate initial and boundary conditions have been specified.

One special case is worth noting. If the situation is static, i.e., if thermal equilibrium has been reached, then \( \partial u/\partial t = 0 \) and the temperature distribution in the rod satisfies

\[
\frac{\partial}{\partial x} \left( K \frac{\partial u}{\partial x} \right) = 0 \hspace{1cm} \text{or (}K\text{ constant)} \hspace{1cm} \frac{\partial^2 u}{\partial x^2} = 0 \hspace{1cm} (15.25)
\]

with, of course, appropriate boundary values at the ends of the rod.

While its deduction is more complicated, Eq. (15.25) has a natural extension to two and three dimensions, in which case we would consider, for example, the evolution of the temperature distribution in a plate or in a three-dimensional object when the initial temperature throughout the object is given and either Dirichlet or Neumann or mixed boundary conditions are specified at all points on its boundary. We would then seek the function \( u(x,y,z,t) = u(r,t) \) giving the temperature at all points in space and time, and the fundamental equation becomes

\[
\nabla \cdot \left( K \nabla u \right) = \rho c \frac{\partial u}{\partial t} \hspace{1cm} \text{or (}K\text{ constant)} \hspace{1cm} \nabla^2 u = \frac{1}{\alpha^2} \frac{\partial u}{\partial t} \hspace{1cm} (15.26)
\]

We thus arrive at the (classical) diffusion equation, the second of the three prototype equations in mathematical physics. The diffusion equation is similar to the wave equation, but differs from it in that the time derivative is only first order.

### 15.1.3 Steady State Heat Flow: The Laplace Equation

The third of the three important equations in mathematical physics is quickly deduced from Eq. (15.26). In two or three dimensions when a steady-state temperature distribution has been reached, \( u \) satisfies the Laplace equation

\[
\nabla^2 u = 0 \hspace{1cm} (15.27)
\]

which, as always, must be supplemented with appropriate boundary conditions before its solution is unique. (This time there will be no initial conditions, since there is no time variable in the picture.)

### 15.1.4 Other Situations

Variants on the three equations we have just deduced appear in many places. For example, Maxwell’s equations for the electrostatic field \( \mathbf{E} \) support the argument

\[
\nabla \times \mathbf{E} = 0 \implies \mathbf{E} = -\nabla V \implies \nabla \cdot (\epsilon \nabla V) = -\rho \hspace{1cm} (15.28)
\]
leading to an equation that determines the electrostatic potential $V$ from the charge density $\rho$ and the dielectric permittivity $\epsilon$. If $\epsilon$ is constant, the equation becomes the Poisson equation,
\[ \nabla^2 V = -\frac{\rho}{\epsilon} \] (15.29)
If, in addition, $\rho = 0$, the equation becomes the Laplace equation
\[ \nabla^2 V = 0 \] (15.30)
for the electrostatic potential.

For another example, suppose we seek a sinusoidal solution to the wave equation, i.e., suppose we seek a solution to Eq. (15.14) of the form
\[ u(x, y, z, t) = \psi(x, y, z) \cos \omega t \]
Then, if $u$ satisfies the wave equation and we set $k^2 = \omega^2/c^2$, $\psi$ will satisfy the equation
\[ \nabla^2 \psi + k^2 \psi = 0 \]
an equation called the Helmholtz equation. In one dimension, the Helmholtz equation becomes
\[ \frac{d^2 \psi}{dx^2} + k^2 \psi = 0 \] (15.31)
which describes standing waves in a string and many other physical situations.

We need not limit our enumeration to equations of importance in classical physics. For example, the time-dependent Schrödinger equation,
\[ -\frac{\hbar^2}{2m} \nabla^2 \psi + V \psi = i\hbar \frac{\partial \psi}{\partial t} \] (15.32)
for the quantum wave function $\psi$ of a particle of mass $m$ in a potential energy $V$ is in some sense a diffusion equation (second order in the space derivatives, first order in the time derivative), though the term $V\psi$ and the imaginary unit $i = \sqrt{-1}$ create significant differences between the two equations.

Beyond Maxwell’s equations and the Schrödinger equation, second-order PDEs can be found in the Dirac equation (relativistic quantum mechanics), the equations of fluid dynamics (the Navier-Stokes’ equations; see Section 15.1.6), the equations of magnetohydrodynamics (MHD, which combine Maxwell’s equations and the Navier-Stokes’ equations), and in many other contexts in classical and contemporary physics.

### 15.1.5 Classification of Second-Order PDEs

In general terms, we have arrived in all cases at second-order, linear, often homogeneous PDEs, equations that we might collectively denote by the symbolism
\[ \mathcal{L}\{u\} = 0 \] (15.33)
where the operator $\mathcal{L}$ symbolizes the (linear) differential operator that defines the equation. The most general example in two variables would have the form
\[ A(x, y) \frac{\partial^2 u}{\partial x^2} + 2B(x, y) \frac{\partial^2 u}{\partial x \partial y} + C(x, y) \frac{\partial^2 u}{\partial y^2} + D(x, y) \frac{\partial u}{\partial x} + E(x, y) \frac{\partial u}{\partial y} + F(x, y) u = 0 \] (15.34)
where $A$, $B$, $C$, $D$, $E$, and $F$ may depend on $x$ and $y$ but do not depend on $u$ or its derivatives. Many, many important equations (though, significantly, not all) fall into this category. The solutions
of those equations that are in this category exhibit the very important property of superposition, namely

\[ \mathcal{L}\{u_1\} = 0, \mathcal{L}\{u_2\} = 0 \implies \mathcal{L}\{au_1 + bu_2\} = a\mathcal{L}\{u_1\} + b\mathcal{L}\{u_2\} = 0 \]  

(15.35)
i.e., any linear combination of two solutions is itself a solution of the differential equation.

As it turns out, second-order, linear PDEs in two variables fall into three distinct categories, depending on the algebraic sign of the quantity \( AC - B^2 \). If, for example, \( AC - B^2 > 0 \), the equation is said to be elliptic; if \( AC - B^2 = 0 \), the equation is parabolic; and if \( AC - B^2 < 0 \), the equation is hyperbolic. When the coefficients actually depend on \( x \) and/or \( y \), an equation may fall into one category for some portions of the region of interest and another category for other portions of that region. Each of the prototype equations we have identified above, however, falls cleanly into one of the three categories. For the wave equation, we note that

\[ \frac{\partial^2 u}{\partial x^2} - \frac{1}{c^2} \frac{\partial^2 u}{\partial t^2} = 0 \implies A = 1, B = 0, C = -\frac{1}{c^2} \implies AC - B^2 = -\frac{1}{c^2} < 0 \]  

(15.36)
and conclude that the wave equation is hyperbolic. For the diffusion equation,

\[ \frac{\partial^2 u}{\partial x^2} - \frac{1}{\alpha^2} \frac{\partial u}{\partial t} = 0 \implies A = 1, B = C = 0 \implies AC - B^2 = 0 \]  

(15.37)
and the diffusion equation is parabolic. Finally, for the Laplace equation,

\[ \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \implies A = C = 1, B = 0 \implies AC - B^2 = 1 \]  

(15.38)
and the Laplace equation is elliptic. In focusing on these three equations, we will therefore be exhibiting techniques that would be applicable to all possible second-order, linear, homogeneous PDEs in two variables.

Note, incidentally, that the variable transformation \( t \to iy \) will convert the wave equation (hyperbolic) into the Laplace equation (elliptic) and remember that this transformation also converts trigonometric functions into hyperbolic functions. Within the framework of complex variable theory, we might therefore expect that solutions to the wave equation and solutions to the Laplace equation could be related to one another (though we shall not pursue that connection in this book).

### 15.1.6 Equations of Fluid Dynamics: Navier-Stokes’ Equations

Let us next work out some of the fundamental equations of fluid dynamics. The state of a fluid at time \( t \) is described by a number of functions, the most important of which are the (vector) velocity field \( \mathbf{v}(\mathbf{r}, t) \), the (scalar) pressure \( p(\mathbf{r}, t) \), and the (scalar) density \( \rho(\mathbf{r}, t) \). In general, all of these quantities are functions of position \( \mathbf{r} \) within the fluid and of time \( t \). Further, \( p \) and \( \rho \) are related by the equation of state of the fluid involved.

Consider a volume—see Fig. 15.5—in the shape of a rectangular parallelepiped with its faces parallel to the coordinate planes. Let the lower left back corner be at \((x, y, z)\) and the upper right front corner be at \((x + \Delta x, y + \Delta y, z + \Delta z)\). We regard the volume as fixed in space, and the fluid as flowing through the volume in a way described by the velocity field already introduced. Further, the fluid in this volume experiences (internal) forces from its contact with the rest of the fluid at the surface of the volume and may also experience (external) forces from things like a nearby earth.\(^3\) The basic equations for fluid motion reflect the conservation of mass (the continuity equation), Newton’s second law, and conservation of linear and angular momentum. We deduce here only the first two of these fundamental relationships.

\(^3\)We assume the fluid is not electrically charged, and we ignore internal forces arising from the gravitational attraction of one portion of the fluid for another. Thus, the only internal forces on one element of the fluid will arise from its direct contact with contiguous elements.
15.1.6.1 Conservation of Mass: The Equation of Continuity

In the time interval from $t$ to $t + \Delta t$, the change in the mass of the fluid in the volume can be calculated in two different ways. First, we focus on the density, concluding that

$$\text{mass added during interval } t \text{ to } t + \Delta t = \left[ \rho(t + \Delta t) - \rho(t) \right] \Delta x \Delta y \Delta z \approx \frac{\partial \rho}{\partial t} \Delta t \Delta x \Delta y \Delta z \quad (15.39)$$

Alternatively, we can calculate this increment in mass from the velocity field. Focus for example on the two sides parallel to the $yz$ plane. On the side at coordinate $x$, all of the fluid in a volume of height $(v_x(x) \Delta t) \Delta y \Delta z$ passes through the surface into the volume in the time interval $\Delta t$. Similarly, on the side at coordinate $x + \Delta x$, all of the fluid in a volume $(v_x(x + \Delta x) \Delta t) \Delta y \Delta z$ passes out of the volume in that same time interval. On the first side, the density of the fluid is $\rho(x)$ while on the second side the density is $\rho(x + \Delta x)$. Thus, the net mass transported into the volume in time $\Delta t$ over these two surfaces by the fluid flow is given by

$$\text{net mass transported into volume through sides parallel to } yz \text{ plane in time } t \text{ to } t + \Delta t = \rho(x) v_x(x) \Delta t \Delta y \Delta z - \rho(x + \Delta x) v_x(x + \Delta x) \Delta t \Delta y \Delta z$$

$$= - \left( \frac{\partial (\rho v_x)}{\partial x} \right) \Delta t \Delta x \Delta y \Delta z \quad (15.40)$$

All quantities in this last expression are evaluated at $x, y, z, t$. We justify ignoring differences in those variables from one point to another by noting that the expression is already first order in $\Delta t$, $\Delta x$, $\Delta y$, and $\Delta z$. Therefore, any variation in $\rho$, $v_x$, $\partial(\rho v_x)/\partial x$, or $\partial\rho/\partial t$ in the (small) interval from $x$ to $x + \Delta x$ would contribute at the strongest to second order in these small quantities—and we will consistently ignore contributions at that level.

Similar expressions apply for the other two pairs of faces (the pair parallel to the $xy$ plane and the pair parallel to the $xz$ plane). In total, the flow of the fluid through the volume transports a net mass given by

$$\text{net mass transported into volume through all sides in time } t \text{ to } t + \Delta t = - \left( \frac{\partial (\rho v_x)}{\partial x} + \frac{\partial (\rho v_y)}{\partial y} + \frac{\partial (\rho v_z)}{\partial z} \right) \Delta t \Delta x \Delta y \Delta z$$

$$= - \nabla \cdot (\rho v) \Delta t \Delta x \Delta y \Delta z \quad (15.41)$$
Figure 15.6: Stresses on the front (dotted) surface of an element in a fluid. (Be sure you visualize the intersection of the three vectors to lie on that front surface.) The signs are defined to give the stresses exerted on an element by the next element in the direction in which the coordinate increases.

The values obtained in Eqs. (15.39) and (15.41) for the mass added in the chosen volume in the time $\Delta t$ must, of course, be the same, and we arrive at the equation of continuity,

$$\nabla \cdot (\rho \mathbf{v}) + \frac{\partial \rho}{\partial t} = 0$$

which expresses the conservation of mass, i.e., the conviction that—however the fluid flows—any change in the mass in a fixed volume can come only by the flow of matter through the surface bounding that volume, i.e., that mass can be neither created nor destroyed in the volume.\(^4\) If the fluid is incompressible (\(\rho\) constant), then the equation of continuity reduces to \(\nabla \cdot \mathbf{v} = 0\).

15.1.6.2 Newton’s Second Law

The volume on which we are focusing our attention experiences forces from the surrounding fluid at its surface. Both normal forces (perpendicular to the surface and related to pressure) and shear forces (tangent to the surface and related to viscosity) may be experienced. All are expressed in terms of stress, normal stress for the first and shear stress for the second, where a stress in general is defined as a force per unit area. For stress, we use the notation $T_{rs}(x,y,z,t)$, where the first subscript conveys the direction of the normal to the surface involved and the second subscript conveys the direction of the stress, i.e.,

$$T_{rs} = \begin{pmatrix} \text{the component in the } s \text{ direction of} \\ \text{the force on a surface of unit area} \\ \text{oriented perpendicular to the } r \text{ direction} \end{pmatrix}$$

Even more specifically, $T_{xx}$ stands for the force in the (positive) $x$ direction (second subscript) on a surface of unit area oriented perpendicular to the $x$ axis (first subscript) and would be a normal stress. Similarly, the symbol $T_{xy}$ stands for the force in the (positive) $y$ direction (second subscript)

\(^4\)The equation, of course, is quite similar to the parallel equation in electrodynamics expressing the conservation of charge.
on a surface of unit area oriented perpendicular to the $x$ axis (first subscript) and would be a shear stress. The three stresses $T_{xx}$, $T_{xy}$, and $T_{xz}$ are shown in Fig. 15.6. As a vector, the total stress on a surface of unit area oriented perpendicular to the $x$ axis would be given by

$$
T_x = T_{xx} \hat{i} + T_{xy} \hat{j} + T_{xz} \hat{k}
$$  \hspace{1cm} (15.44)

Note particularly that the stresses shown are the stresses on the indicated side of the illustrated element arising from its contact with the adjacent portion of the fluid in the direction in which the $x$ coordinate increases. The stresses that the illustrated element exerts on its neighbor in the positive $x$ direction will, via Newton’s third law, be equal in magnitude but opposite in direction to those shown. In particular, then, the net force arising from internal stresses on the front and back surfaces of the illustrated element would be given by

$$
F_x = \left[ \left( T_{xx}(x + \Delta x, y, z) - T_{xx}(x, y, z) \right) \hat{i} + \left( T_{xy}(x + \Delta x, y, z) - T_{xy}(x, y, z) \right) \hat{j} + \left( T_{xz}(x + \Delta x, y, z) - T_{xz}(x, y, z) \right) \hat{k} \right] \Delta y \Delta z
$$

$$
= \left( \frac{\partial T_{xx}}{\partial x} \hat{i} + \frac{\partial T_{xy}}{\partial x} \hat{j} + \frac{\partial T_{xz}}{\partial x} \hat{k} \right) \Delta x \Delta y \Delta z
$$  \hspace{1cm} (15.45)

Here, we can evaluate the tensions at $y, z$ because variation of $y$ and $z$ over the front and back surfaces will contribute only to second order in $\Delta y$ and $\Delta z$—and we are ignoring terms beyond first order. Further, the derivatives can all be evaluated at argument $(x, y, z)$, since the variation between the front and back surfaces will contribute only to second order in $\Delta x$. For clarity in the final result, those arguments have been omitted.\(^5\)

Only the subscripts and the variable with respect to which stresses are differentiated must be adjusted to yield expressions for the forces arising from internal stresses on the other two pairs of surfaces on the element in question. We find that

$$
F_y = \left( \frac{\partial T_{yx}}{\partial y} \hat{i} + \frac{\partial T_{yy}}{\partial y} \hat{j} + \frac{\partial T_{yz}}{\partial y} \hat{k} \right) \Delta x \Delta y \Delta z
$$  \hspace{1cm} (15.46)

and

$$
F_z = \left( \frac{\partial T_{zx}}{\partial z} \hat{i} + \frac{\partial T_{zy}}{\partial z} \hat{j} + \frac{\partial T_{zz}}{\partial z} \hat{k} \right) \Delta x \Delta y \Delta z
$$  \hspace{1cm} (15.47)

Finally, bringing together all of the $x$ components of the forces arising from internal stress on all surfaces of the element on which our attention is focused and adding a possible external force\(^6\) $\rho f_x \Delta x \Delta y \Delta z$, we conclude that the $x$ component of the net force on the element in question would be given by

$$
F_x = \left( \rho f_x + \frac{\partial T_{xx}}{\partial x} + \frac{\partial T_{yx}}{\partial y} + \frac{\partial T_{xz}}{\partial z} \right) \Delta x \Delta y \Delta z
$$  \hspace{1cm} (15.48)

According to Newton’s second law, this force must, of course, also be given as the product of the mass of the element times the $x$ component of its acceleration, i.e., by $(\rho \Delta x \Delta y \Delta z) a_x$. Thus, we have for the $x$ component of the equation of motion the result

$$
\rho a_x = \rho f_x + \frac{\partial T_{xx}}{\partial x} + \frac{\partial T_{yx}}{\partial y} + \frac{\partial T_{xz}}{\partial z}
$$  \hspace{1cm} (15.49)

Similar equations could be deduced for the $y$ and $z$ components of the equation of motion for the fluid.

\(^5\)It is extremely easy to become very confused about these signs. Think about them several times.

\(^6\)For convenience, we define $f_x$ to be a force per unit mass.
Evaluating the acceleration, however, introduces a subtle complication, since \( a_x \neq \frac{\partial v_x}{\partial t} \).\footnote{This derivative is the rate of change of the velocity field at a fixed position in space. Unfortunately, at the end of the time interval, the element of the fluid on which the force acts is at a new position in the fluid. The quantity that figures in the definition of the acceleration we want is the change in the velocity of a particular element of the fluid that moves with the fluid; that quantity is given by} This derivative is the rate of change of the velocity field at a fixed position in space. Unfortunately, at the end of the time interval, the element of the fluid on which the force acts is at a new position in the fluid. The quantity that figures in the definition of the acceleration we want is the change in the velocity of a particular element of the fluid that moves with the fluid; that quantity is given by

\[
\Delta v = v(x + v_x \Delta t, y + v_y \Delta t, z + v_z \Delta t, t + \Delta t) - v(x, y, z, t)
\]

which implies that the acceleration we really want is given by

\[
a = \frac{\Delta v}{\Delta t} = v_x \frac{\partial v}{\partial x} + v_y \frac{\partial v}{\partial y} + v_z \frac{\partial v}{\partial z} + \frac{\partial v}{\partial t} = \frac{\partial v}{\partial t} + (v \cdot \nabla)v \tag{15.51}
\]

a derivative that is sometimes said to “follow the fluid” and is sometimes called the \textit{substantial derivative}. Thus, the translation of the \( x \) component of Newton’s second law into the vocabulary used to describe the state of a fluid is

\[
\rho \left( \frac{\partial}{\partial t} + v \cdot \nabla \right) v_x = \rho f_x + \frac{\partial T_{xx}}{\partial x} + \frac{\partial T_{yx}}{\partial y} + \frac{\partial T_{zx}}{\partial z} \tag{15.52}
\]

The \( y \) and \( z \) components have similar statements.\footnote{To write a single equation combining the \( x \), \( y \), and \( z \) components, we would have to introduce the notation of tensors. Given our limited use of these relationships, we have little motivation to take that step.} Equation (15.52) and its \( y \) and \( z \) counterparts constitute a set of complicated, non-linear, coupled PDEs. Further, they are not by themselves complete, since they involve the stresses (which are not among the official quantities used to describe the state of the fluid). The equations (including the continuity equation) have too many unknowns (\( \rho, v \), and the stresses). We need some further relationships, particularly relationships linking stresses to fluid velocities (and involving new fluid parameters like viscosity). For example, in some simple cases, we can sometimes suppose that the shear stress in one direction (say \( x \)) on a surface perpendicular to another direction (say \( y \)) is related to the rate at which the component of the velocity in the first direction changes with displacement in the second direction, i.e., \( T_{yx} = \mu \frac{\partial v_x}{\partial y} \), where \( \mu \) is the viscosity of the fluid (which in effect is defined by this relationship).\footnote{A fluid described by this relationship is said to be a \textit{Newtonian} fluid. The relationship is an approximation, but fortunately—there are several fluids to which it—and its generalizations to 2D and 3D flow—seem to be accurately applicable.}

15.1.6.3 A Special Case: Non-viscous Flow

If the fluid of interest has zero viscosity, then there can be no shear forces and all the off-diagonal elements of the stress tensor \( T \) will be zero. Further, in most instances, the on-diagonal elements will all be equal to the negative of the pressure. That is, \( T_{xx} = T_{yy} = T_{zz} = -p \) and \( T_{xz} = T_{xy} = \ldots = 0 \). In that case, Eq. (15.52) reduces to

\[
\rho \left( \frac{\partial}{\partial t} + v \cdot \nabla \right) v_x = \rho f_x - \frac{\partial p}{\partial x} \tag{15.53}
\]

Combined in vector notation with the other two components, the basic equation for \textit{non-viscous} flow then is that

\[
\rho \left( \frac{\partial}{\partial t} + v \cdot \nabla \right) v = \rho f - \nabla p \tag{15.54}
\]

Once the external force and the equation of state relating \( \rho \) and \( p \) have been determined, this equation provides the starting point for solving many problems in non-viscous fluid flow.
15.1.6.4 A Second Special Case: Sound Waves

Suppose the flow of interest is 1D, so \( v_y \) and \( v_z \) are both zero and \( v_x \) depends only on \( x \) and \( t \). Further, suppose that external forces are absent and that the flow is inviscid, i.e., that the viscosity is zero and hence that shear stresses are zero \( (T_{yx} = 0 \text{ and } T_{zx} = 0) \). Then Eq. (15.42) expressing conservation of mass and Eq. (15.52) expressing Newton’s second law become

\[
\frac{\partial}{\partial x} (\rho v_x) = - \frac{\partial \rho}{\partial x}; \quad \rho \frac{\partial v_x}{\partial t} + v_x \frac{\partial v_x}{\partial x} = - \frac{\partial p}{\partial x} \tag{15.55}
\]

(We have recognized that \( T_{xx} = -p \).) Further, suppose that the velocity of the fluid is small, so we can ignore the non-linear term and reduce these equations to

\[
\frac{\partial}{\partial x} (\rho v_x) = - \frac{\partial \rho}{\partial t}; \quad \rho \frac{\partial v_x}{\partial t} = - K \frac{\partial \rho}{\partial x} \tag{15.56}
\]

Next, let us suppose that \( p = p(\rho) \), i.e., that the pressure in the fluid and its density are functionally related. Then,

\[
\frac{\partial p}{\partial x} = \frac{dp}{d\rho} \frac{\partial \rho}{\partial x} = K \frac{\partial \rho}{\partial x} \tag{15.57}
\]

where \( K = dp/d\rho \), which will in general depend on \( \rho \), is a property of the fluid characterizing the extent to which small changes in applied pressure are determined from small changes in density. The equations now become

\[
\frac{\partial}{\partial x} (\rho v_x) = - \frac{\partial \rho}{\partial t}; \quad \rho \frac{\partial v_x}{\partial t} = - K_0 \frac{\partial \rho}{\partial x} \tag{15.58}
\]

Finally, we suppose that the pressure departs only slightly from its nominal equilibrium value \( \rho_0 \) and that the velocity in equilibrium is zero (the fluid is quiescent), i.e., we set

\[
\rho = \rho_0 + \delta \rho(x,t); \quad v_x = \delta v_x(x,t) \tag{15.59}
\]

Then, ignoring all but the linear terms in small quantities, we find that the equations now are

\[
\rho_0 \frac{\partial \delta v_x}{\partial x} = - \frac{\partial \delta \rho}{\partial t}; \quad \rho_0 \frac{\partial \delta v_x}{\partial t} = - K_0 \frac{\partial \delta \rho}{\partial x} \tag{15.60}
\]

where \( K_0 = K(\rho_0) \) is the value of \( K \) associated with the equilibrium density \( \rho_0 \)—and is constant. Differentiating the first of these with respect to \( t \) and the second with respect to \( x \), we find that

\[
\rho_0 \frac{\partial^2 \delta v_x}{\partial t \partial x} = \frac{\partial^2 \delta \rho}{\partial t^2}; \quad \rho_0 \frac{\partial^2 \delta v_x}{\partial x \partial t} = - K_0 \frac{\partial^2 \delta \rho}{\partial x^2} \tag{15.61}
\]

Since the mixed second-partial derivatives are equal, we then conclude that

\[
\frac{\partial^2 \delta \rho}{\partial t^2} = K_0 \frac{\partial^2 \delta \rho}{\partial x^2} \tag{15.62}
\]

That is, the density fluctuations \( \delta \rho \) in this system satisfy the wave equation. In effect, we have discovered that, among the motions possible in this medium, there is a longitudinal wave that propagates with speed \( v = \sqrt{K_0} \). [See Eq. (15.13) and footnote 2.]

As an aside, note that, for an ideal gas, \( pV = nRT \) or \( p = nRT/V = MRT/(V_m n) \), where \( M \) is the mass of the sample of gas and \( m_n \) is the mass of the gas per mole. Further, when the gas undergoes an adiabatic process, as it does when it supports a sound wave, \( pV^\gamma \), where \( \gamma \) is the ratio of the specific heat at constant pressure to that at constant volume, is constant. Thus, since \( pV^\gamma = p_0 V_0^\gamma \) and \( V = M/\rho \), we find that

\[
p \left( \frac{M}{\rho} \right)^\gamma = p_0 V_0^\gamma \quad \Rightarrow \quad p = \frac{p_0 V_0^\gamma}{M^\gamma \rho^\gamma} \quad \Rightarrow \quad K = \frac{dp}{d\rho} = \gamma \frac{p_0 V_0^\gamma}{M^\gamma \rho^{\gamma-1}} \tag{15.63}
\]
or, evaluating at $\rho = \rho_0$ and noting the relationship $V_0 = M/\rho_0$, that

$$K_0 = \frac{dp}{d\rho_0} = \gamma \frac{\rho_0 V_0^\gamma}{M^\gamma} \rho_0^{\gamma-1} = \gamma \frac{\rho_0 M^\gamma}{M^\gamma \rho_0^{\gamma-1}} = \gamma \frac{\rho_0}{\rho_0} = \gamma \frac{RT_0}{m_n}$$

and we have found $K_0$ for this simple system. Note that $K_0$ is temperature dependent. Further, with this value for $K_0$, we can predict that the speed of sound in an ideal gas at temperature $T_0$ is given by $v = \sqrt{K_0} = \sqrt{\gamma RT_0/m_n}$. In particular, we conclude that the speed of sound in a gas increases as the square root of the (absolute) temperature. With $R = 8.31 \text{ N m}/(\text{mol K})$ and—for air—$m_n = 0.0288 \text{ kg}$ and $\gamma = 7/5 = 1.4$, we find at room temperature (70 °F = 21 °C = 294 K) that

$$v = \sqrt{\gamma RT_0} = \sqrt{1.4(8.31)(294)/0.0288} \text{ m/s} = 344 \text{ m/s} \quad (15.65)$$

A further interesting connection appears if we remember from the equipartition theorem of kinetic theory that the rms speed $v_{rms}$ of molecules of mass $m$ in a gas at absolute temperature $T_0$ satisfies $\frac{1}{2}mv_{rms}^2 = \frac{3}{2}kT_0$ or, on multiplying by Avogadro’s number, $\frac{1}{2}m_n v_{rms}^2 = \frac{3}{2}RT_0$. We conclude that

$$v_{rms} = \sqrt{\frac{3RT_0}{m_n}} \implies v = \sqrt{\frac{\gamma RT_0}{m_n}} = \sqrt{\frac{\gamma}{3} \frac{RT_0}{m_n}} = \sqrt{\frac{\gamma}{3} v_{rms}} \quad (15.66)$$

We thus find that the speed of sound in a gas is a simple multiply—0.683 in the case of air—of the root mean square speed of the molecules of the gas and that both increase as the square root of the temperature!

15.1.7 A General 1D Equation

While a differential equation in one dimension is, of course, an ODE, not a PDE, description and illustration of the essential approaches underlying both finite difference methods and finite element methods are simplest in one dimension. We elect to begin there, seeking as an example to solve the general linear, second-order, self-adjointootnote{The word self-adjoint characterizes an equation in which—see the second form in Eq. (15.67)—the coefficient of the first derivative term is the derivative of the coefficient of the second derivative term. While this requirement appears to be restrictive, it turns out—see Exercise 15.3—that any linear, second-order equation can with an appropriately chosen integrating factor be cast in self-adjoint form, so the restriction is only apparent, not real.} inhomogeneous equation

$$-\frac{d}{dx} \left( \alpha(x) \frac{d\varphi(x)}{dx} \right) + \beta(x) \varphi(x) = f(x) \quad \text{or} \quad -\alpha(x) \frac{d^2\varphi(x)}{dx^2} - \alpha'(x) \frac{d\varphi(x)}{dx} + \beta(x) \varphi(x) = f(x) \quad (15.67)$$

in the interval $0 \leq x \leq L$. Here, $\varphi(x)$ is an unknown function, $\alpha(x)$ and $\beta(x)$ are parameters related to the physical properties of the problem (and may be functions of $x$), $\alpha'(x) = d\alpha(x)/dx$, and $f(x)$ is a forcing or source function (inhomogeneity). With appropriate choices of $\alpha$, $\beta$, and $f$, this equation can be reduced to the one-dimensional versions of several of the equations we have developed in the previous subsections.

To be complete, we shall suppose we are dealing with a boundary value problem, taking the desired boundary conditions to be given by

$$\varphi(0) = p \quad (15.68)$$

and

$$\left[ \alpha(x) \frac{d\varphi(x)}{dx} + \gamma \varphi(x) \right]_{x=L} = q \quad (15.69)$$

where $p$, $\gamma$, and $q$ are known constants. Together, these conditions will allow us to demonstrate how to treat each of the three kinds of boundary conditions. The condition at $x = 0$ is a Dirichlet boundary condition or boundary condition of the first kind; the condition at $x = L$ is a mixed boundary condition or boundary condition of the third kind. When $\gamma = 0$ in this second condition, the condition reduces to a Neumann condition or boundary condition of the second kind.
15.1.8 A General 2D Equation

As a simple—and general—two-dimensional example, we will illustrate the application of finite difference and finite element analysis to solve an equation of the general form

\[-\frac{\partial}{\partial x} \left( \alpha_x(x, y) \frac{\partial \varphi}{\partial x} \right) - \frac{\partial}{\partial y} \left( \alpha_y(x, y) \frac{\partial \varphi}{\partial y} \right) + \beta(x, y) \varphi = f(x, y) \]  \hspace{1cm} (15.70)

where \(\alpha_x, \alpha_y\) and \(\beta\) are known quantities—possibly constants; possibly functions of \(x\) and \(y\)—and \(f\) also possibly constant; possibly a function of \(x\) and \(y\)—is a driving term (inhomogeneity). While this equation is presented in its most general form so that the results of our discussion can be applied to a variety of different physical problems, appropriate restrictions of \(\alpha_x, \alpha_y, \beta,\) and \(f\) will reduce it to the Laplace, Poisson, or Helmholtz equation. To complete the statement of the problem, we suppose that a solution is to be found subject to the Dirichlet conditions

\[\varphi = p \quad \text{(on } \Gamma_1)\]  \hspace{1cm} (15.71)

on the portion \(\Gamma_1\) of the boundary and the Neumann conditions

\[\left( \alpha_x \frac{\partial \varphi}{\partial x} \hat{i} + \alpha_y \frac{\partial \varphi}{\partial y} \hat{j} \right) \bullet \mathbf{n} = q \quad \text{(on } \Gamma_2)\]  \hspace{1cm} (15.72)

on the remainder \(\Gamma_2\) of the boundary. Here, \(p\) and \(q\) are quantities defined on the boundary, and \(\mathbf{n}\) is a unit vector perpendicular to the boundary and directed outward from the perspective of a viewer in the region in which a solution is sought.

15.2 Finite Difference Methods (FDMs) in One Dimension

In the finite difference approach to the one-dimensional problem laid out in Section 15.1.7, we begin by dividing the interval \(0 \leq x \leq L\) into \(N\) segments, each of length \(\Delta x = L/N\), so that the \(i\)-th node \((i = 0, 1, 2, \ldots, N)\) has \(x\) coordinate \(x_i = i \Delta x\). In particular \(x_0 = 0\) and \(x_N = L\). Then, we evaluate Eq. (15.67) at the point \(x_i\) to find that

\[-\alpha_i \frac{d^2 \varphi(x)}{dx^2} \bigg|_{x_i} - \alpha'_i \frac{d \varphi(x)}{dx} \bigg|_{x_i} + \beta_i \varphi_i = f_i \]  \hspace{1cm} (15.73)

where \(\alpha_i = \alpha(x_i), \alpha'_i = \alpha'(x_i), \beta_i = \beta(x_i),\) and \(f_i = f(x_i)\) are all known and \(\varphi_i = \varphi(x_i)\) is to be found. Next, we approximate the derivatives by invoking finite differences. We illustrate only the most common of several possible ways to achieve that objective. In terms of the quantities \(\varphi_i,\) we might, for example, write the first derivative at \(x_i\) in any of the ways

\[\frac{d \varphi(x)}{dx} \bigg|_{x_i} \approx \frac{\varphi_{i+1} - \varphi_i}{\Delta x} \quad \text{or} \quad \frac{d \varphi(x)}{dx} \bigg|_{x_i} \approx \frac{\varphi_i - \varphi_{i-1}}{\Delta x} \quad \text{or} \quad \frac{d \varphi(x)}{dx} \bigg|_{x_i} \approx \frac{\varphi_{i+1} - 2\varphi_i + \varphi_{i-1}}{2\Delta x} \]  \hspace{1cm} (15.74)

or probably in other ways as well (see Exercise 15.25). The first of these approximations involves a forward difference, the second involves a backward difference, and the third involves a central difference. Each is correct, though they are not all equally convenient or useful—nor is any single one always the most appropriate or convenient. For the present example, we choose the central difference to approximate the first derivative in Eq. (15.73). To find an approximation for the second derivative, however, we use both the forward and the backward approximations for the first derivative, writing that

\[\frac{d^2 \varphi(x)}{dx^2} \bigg|_{x_i} \approx \frac{d \varphi(x)}{dx} \bigg|_{x_i + \Delta x/2} - \frac{d \varphi(x)}{dx} \bigg|_{x_i - \Delta x/2} = \frac{\varphi_{i+1} - \varphi_i}{\Delta x} - \frac{\varphi_i - \varphi_{i-1}}{\Delta x} = \frac{\varphi_{i+1} - 2\varphi_i + \varphi_{i-1}}{\Delta x^2} \]  \hspace{1cm} (15.75)

10 Note that the central difference formula is the average of the forward and backward formulae.
Here, we have recognized that the forward (backward) difference approximation to the derivative at \( x_i \) is a central difference approximation to the derivative at \( x_i + \frac{1}{2} \Delta x \) \((x_i - \frac{1}{2} \Delta x)\), and we have taken the second derivative at \( x_i \) to be approximated by the difference of these two approximations to the first derivative divided by the separation of the points at which the two first derivatives are evaluated.\(^{13}\) Finally, we substitute the approximation of Eq. (15.75) and the central difference approximation of Eq. (15.74) into Eq. (15.73) to find that

\[
-\alpha_i \left( \frac{\varphi_{i+1} - 2\varphi_i + \varphi_{i-1}}{\Delta x^2} \right) - \alpha_i' \left( \frac{\varphi_{i+1} - \varphi_{i-1}}{2\Delta x} \right) + \beta_i \varphi_i = f_i
\]

(15.76)
or, on multiplying by \( \Delta x^2 \) and collecting terms with the same index on \( \varphi \), that

\[
\left( -\alpha_i + \frac{\alpha_i' \Delta x}{2} \right) \varphi_{i-1} + \left( 2\alpha_i + \beta_i \Delta x^2 \right) \varphi_i + \left( -\alpha_i - \frac{\alpha_i' \Delta x}{2} \right) \varphi_{i+1} = f_i \Delta x^2
\]

(15.77)

Since \( i \) can assume any value in the interval \( 0 \leq i \leq N \), we thus have \( N + 1 \) equations, just the right number to determine the \( N + 1 \) unknowns \( \varphi_i \).

The conclusion of the last sentence of the previous paragraph, however, is premature. Unfortunately, when evaluated at \( i = 0 \) or \( i = N \), Eq. (15.77) makes reference to \( \varphi_{-1} \) or \( \varphi_{N+1} \), each of which is outside the domain of the problem! Thus, we really have \( N + 3 \) unknowns. The additional information we need lies in the boundary conditions, though the precise way in which these conditions resolve this problem depends on the type of boundary condition. For the Dirichlet condition of Eq. (15.68), we abandon the first equation \( (i = 0) \) altogether, replacing it with the equation prescribed by the boundary conditions, namely

\[
\varphi_0 = p
\]

(15.78)

For the mixed boundary condition of Eq. (15.69), we use a central difference approximation to the derivative in the boundary condition, finding that

\[
\alpha_N \left( \frac{\varphi_{N+1} - \varphi_{N-1}}{2\Delta x} \right) + \gamma \varphi_N = q \quad \implies \quad \varphi_{N+1} = \varphi_{N-1} + \frac{2 \Delta x}{\alpha_N} (q - \gamma \varphi_N)
\]

(15.79)

where \( \alpha_N = \alpha(x_N) = \alpha(L) \). Then, we write out the last equation \( (i = N) \) and substitute from Eq. (15.79) to eliminate \( \varphi_{N+1} \), finding ultimately that

\[
-2 \alpha_N \varphi_{N-1} + \left[ 2 \left( \alpha_N + \gamma \Delta x \right) + \left( \beta_N + \frac{\gamma \alpha_N'}{\alpha_N} \right) \Delta x^2 \right] \varphi_N = \left( f_N + \frac{\alpha_N' q}{\alpha_N} \right) \Delta x^2 + 2q \Delta x
\]

(15.80)

With these resolutions of the first and last equations, we arrive at the now fully defined and complete set

\[
\varphi_0 = p
\]

\[
\left( -\alpha_i + \frac{\alpha_i' \Delta x}{2} \right) \varphi_{i-1} + \left( 2\alpha_i + \beta_i \Delta x^2 \right) \varphi_i + \left( -\alpha_i - \frac{\alpha_i' \Delta x}{2} \right) \varphi_{i+1} = f_i \Delta x^2, \quad 1 \leq i \leq N - 1
\]

\[
-2 \alpha_N \varphi_{N-1} + \left[ 2 \left( \alpha_N + \gamma \Delta x \right) + \left( \beta_N + \frac{\gamma \alpha_N'}{\alpha_N} \right) \Delta x^2 \right] \varphi_N = \left( f_N + \frac{\alpha_N' q}{\alpha_N} \right) \Delta x^2 + 2q \Delta x
\]

(15.81)
of \( N + 1 \) equations for the \( N + 1 \) unknowns \( \varphi_0, \varphi_1, \ldots, \varphi_N \). Remember that, for definiteness, we have chosen to use Dirichlet boundary conditions at \( x = 0 \) and mixed boundary conditions at \( x = L \). In other situations, the conditions at the two ends might both be of the same type, or they could each

\(^{13}\)You may have to read this sentence several times. The appearance of the binomial coefficients \((1, 2, 1)\), as in \((a + b)^2 = a^2 + 2ab + b^2\), is worth noting.
be of a type different from those here illustrated. In any case, we have illustrated how to address all three possible types in the specific choices made.

In effect, we have deduced a set of linear algebraic equations to be solved for the unknowns \( \varphi_0, \varphi_1, \ldots, \varphi_N \). For visualization, we note that the set can be seen in matrix form. Further, since each equation involves no more than three consecutive indices, the matrix of the coefficients is tridiagonal though not necessarily symmetric. For example, if \( N = 10 \), the matrix version of these equations would have the general form

\[
\begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
? & ? & ? & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & ? & ? & ? & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & ? & ? & ? & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & ? & ? & ? & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & ? & ? & ? & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & ? & ? & ? & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & ? & ? & ? & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & ? & ? & ? \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & ? & ? \\
\end{bmatrix}
\begin{bmatrix}
\varphi_0 \\
\varphi_1 \\
\varphi_2 \\
\varphi_3 \\
\varphi_4 \\
\varphi_5 \\
\varphi_6 \\
\varphi_7 \\
\varphi_8 \\
\varphi_9 \\
\varphi_{10}
\end{bmatrix}
= 
\begin{bmatrix}
p \\
? \\
? \\
? \\
? \\
? \\
? \\
? \\
? \\
? \\
? \\
\end{bmatrix}
\]

In this display, only those elements marked with "?" can differ from zero (though some of those may be zero as well), and every non-zero element except those in the column of \( \varphi_i \)'s is known at the outset.

Two cases are important. In the first case, the equations are inhomogeneous and the determinant of the coefficient matrix is not zero; and the equations have a unique solution that will approximate the solution to the original boundary value problem. In the second case, the equations are homogeneous, and we hope that the the coefficient matrix contains a parameter that can be adjusted to make the determinant of that matrix zero; and our problem involves seeking the eigenvalues and eigenvectors of the coefficient matrix; we expect to find several solutions, one corresponding to each eigenvalue of the coefficient matrix.\(^{12}\)

Finite difference approaches to PDEs are, of course, subject to error. In general, these errors fall into three distinct and independent categories. In the first instance, we have replaced a continuous variable by a discrete set of values of that variable, i.e., we have discretized the independent variable. In so doing, we have rendered the solution vulnerable to discretization error, an error that will be reduced as the separation of the discrete values is made smaller. From that perspective alone, we would want to make that separation as small as possible, the smaller the better. Unfortunately, as we make the separation smaller, we also increase the computational labor of generating the solution and hence the time required to complete the solution. What’s worse, increasing the computational labor also increases the likelihood that errors will be generated by computer roundoff, which arises because computers retain floating point numbers only to finite precision. We thus must seek a compromise: We want to use a grid with small intervals so as to reduce discretization error but we can’t make the grid too small because roundoff errors may then become significant. Fortunately, in most cases, we will be able to find a grid that is simultaneously fine enough to render discretization error of little consequence and large enough to keep roundoff errors at bay. The standard assessment of these issues involves solving the problem twice, once with a coarser grid and then again with a finer grid. Until roundoff becomes a problem, we can (almost) always safely assume that the solution on the finer grid is more accurate than that on the coarser grid, and a comparison of the two solutions will provide some indication of the accuracy of the solution.

Sometimes, we will use a direct algorithm to solve the discretized equations that convey the solution, and the third type of error will not be present. When an interactive algorithm is used,

\(^{12}\)Two other cases exist but are of no significance. The equations might be inhomogeneous and the determinant of the coefficient matrix zero, in which case no solution exists, or the equations might be homogeneous and the determinant of the coefficient matrix non-zero, in which case only the trivial solution (all \( \varphi \)'s zero) exists.
however, we need be concerned not only about discretization and roundoff errors; we must also assess convergence error. In those cases, we are using a method that may or may not converge to the solution of the approximate equations. We must therefore worry not only about the extent to which the solution to the equations we are solving actually represents the solution to the original problem (discretization and roundoff errors) but also about the extent to which the solution we ultimately deduce is actually a solution to the approximating equations (convergence error). Again, we assess the accuracy of the solution to the approximating equations by comparing successive iterates that we hope are converging on that solution. That hope will be more or less justified depending on the rapidity of the apparent convergence.

We can carry our analysis no further without selecting a particular programming language in which to implement explicit coding. That task is undertaken in the next section(s).

15.8 Using C to Solve 1D PDEs via an FDM

15.8.1 A General Coding Using LAPACK

The final step is to solve Eq. (15.82) for the unknowns $\varphi_0, \varphi_1, \ldots, \varphi_N$. The numerical operations associated with constructing and then solving Eq. (15.82) are clearly a job for a computer, especially when the problem to be solved involves more than a very few simultaneous equations. A program to construct Eq. (15.82) and then solve the resulting system for $\varphi$ at each node would begin by asking the user to input the values of the various parameters involved in the problem. In all cases, the parameters $p$, $\gamma$, and $q$, which relate to the boundary conditions, are constants. In general, the quantities $\alpha$, $\alpha'$, and $\beta$, which relate to the physical system involved, and $f$, which represents a source or excitation function, will depend on $x$. For simplicity in illustration, however, we will suppose these four quantities to be constants as well. Indeed, if $\alpha$ is constant, then $\alpha'$ is zero, and we will simply leave out terms multiplied by $\alpha'$ when we construct the equations.

The first segment of the program we wish to write will, then, request input of all the constants needed in the remainder of the program. Appropriate statements in C are

```c
#include <stdio.h>
#include <math.h>
float alpha, beta, f, L, p, gamma, q;
int N;

printf( "Enter number of segments (N): " ) ; scanf( "%d", &N );
printf( "Enter alpha: " ) ; scanf( "%f", &alpha );
printf( "Enter beta: " ) ; scanf( "%f", &beta );
printf( "Enter f: " ) ; scanf( "%f", &f );
printf( "Enter L: " ) ; scanf( "%f", &L );
printf( "Enter p: " ) ; scanf( "%f", &p );
printf( "Enter gamma: " ) ; scanf( "%f", &gamma );
printf( "Enter q: " ) ; scanf( "%f", &q );
```

To provide the flexibility to specify the refinement of elements without having to rewrite the program, we elect to dimension several needed arrays at execution time. To allocate memory at

---

13LAPACK is a large FORTRAN package of routines that implement many algorithms in linear algebra, among them routines for solving a wide variety of systems of linear algebraic equations. Information about acquiring components of this package are provided in Appendix Z. We illustrate in this section how to call FORTRAN routines from a C program.

14If $\alpha$, $\alpha'$, $\beta$, and $f$ are not constant, each of these quantities would have to be represented by an $N + 1$ element vector and values would have to be given—or computed—for all of those elements.
CHAPTER 15. SOLVING PARTIAL DIFFERENTIAL EQUATIONS

execution time, we must invoke the C function malloc which is defined in the include file stdlib.h. Then, once we know the required sizes of the arrays, we allocate the necessary space explicitly.\(^{15}\) The statements

```c
#include <stdlib.h>  /* Needed for the function malloc */
float *x, dx, dx2;  /* x is declared as a pointer */
x = (float*)malloc((N+1)*sizeof(float));
int I;

dx = L/N; dx2 = pow(dx, 2);
for(I=0; I<=N; I++) x[I] = dx*I;
```

(1) include the needed file library file, (2) declare the variable \(x\) as a pointer so we can subsequently specify its size, (3) declare additional variables, (4) allocate the required memory for \(x\), and (5) calculate the size of each segment \(dx\), the square of that size \(dx^2\), and the values of \(x\) at which (equally spaced) nodes will be placed in the interval \(0 \leq x \leq L\).

For the final solution of this system of equations, we elect to use the LAPACK routine \(\text{sgtsv.f}\).\(^{16}\) which is a single-precision (s) routine for the solving general\(^{17}\) (g), tridiagonal (t), linear algebraic equations. Among its inputs, that routine requires vectors inhomogeneous, the diagonal elements of the coefficient matrix, the superdiagonal elements of that matrix, and the subdiagonal elements of that matrix, respectively, all of which must be declared as pointers so we can then allocate appropriate memory once the required size is known. Thus, we add the statements

```c
float *inhomo, *supdiag, *diag, *subdiag;
inhomo = (float*)malloc( (n+1)*sizeof(float) );
supdiag = (float*)malloc( n*sizeof(float) );
diag = (float*)malloc( (n+1)*sizeof(float) );
subdiag = (float*)malloc( n*sizeof(float) );

for(I=0; I<=N; I++) {
    inhomoi[I] = f*dx2;
    diag[I] = 2.0*alpha + beta*dx2;
};
inhomeo[0] = p;
inhomeo[N] = inhomeo[N] + 2.0*q*dx;
diag[0] = 1.0;
diag[N] = diag[N] + 2.0*gamma*dx;

for(I=0; I<=N-1; I++) {
    supdiag[I] = -alpha;
    subdiag[I] = -alpha;
};
supdiag[0] = 0.0;
subdiag[N-1] = -2.0*alpha;
```

to declare the necessary vectors and assign values reflecting Eqs. (15.81) and (15.82). (Note that, as required by \(\text{sgtsv}\), each of \(\text{supdiag}\) and \(\text{subdiag}\) is one element shorter than \(\text{diag}\).)

\(^{15}\)Technically, once we no longer need the dynamically allocated variable, we should us the function free() to free up the allocated space for other uses. Since we here are not taxing memory and since the space will be freed as soon as the program returns control to the operating system, we need not free the space explicitly in the program.

\(^{16}\)This FORTRAN function can be called from a C program in a way to be described before the end of this subsection.

\(^{17}\)Systems whose coefficient matrix is not necessarily symmetric.
We are now ready to invoke the LAPACK tridiagonal FORTRAN solver `sgtsv` to generate the solution from the several quantities we have now evaluated. We invoke the statements

```c
int NN, info;
NN = N + 1; I = 1;
sgtsv_( &NN, &I, subdiag, diag, supdiag, inhomu, &NN, &info );
```

to declare two additional variables, establish particular input values and solve the system of equations. Here

- The first argument, `NN`, on input is the number of equations; it is not changed on output.
- The second argument, `I`, on input is the number of columns in `inhomo`; it is not changed on output.
- The third argument, `subdiag`, on input is the vector of values on the subdiagonal of the coefficient matrix; on output it contains pieces of the LU decomposition of the coefficient matrix. The variable is already a pointer so needs no ampersand.
- The fourth argument, `diag`, on input is the vector of values on the diagonal of the coefficient matrix; on output it contains additional pieces of the LU decomposition of the coefficient matrix. The variable is already a pointer so needs no ampersand.
- The fifth argument, `supdiag`, on input is the vector of values on the superdiagonal of the coefficient matrix; on output it contains additional pieces of the LU decomposition of the coefficient matrix. The variable is already a pointer so needs no ampersand.
- The sixth argument `inhomo` on input is the array of inhomogeneities; on output it is the array of solutions. The variable is already a pointer so needs no ampersand.
- The seventh argument, `NN`, on input is the number of rows in `inhomo`; it is not changed on output.
- The eighth argument, `info`, is not used on input; on output its value conveys whether or not the solution is successful.

More detailed information about these variables is compiled in the comments at the beginning of the LAPACK file `sgtsv.f`.

Finally, to write the size of the vector containing the solution and also the solution itself into a file for examination by other programs, we add the statements

```c
FILE *fptr;
fptr = fopen( "fdm1dla_c.dat", "w" );
fprintf( fptr, "%d\n", N + 1 );
for( I=0; I<=N; I++ ) fprintf( fptr, "%7.2f %10.4f\n", x[I], inhomu[I] );
fclose( fptr );
```

A more fully commented listing of the complete program `fdm1dla.c`, is presented in Appendix 15.A.6. The program itself can be copied from the directory `$HEAD/cc`.

We are at long last ready to compile and link this program with the needed LAPACK routines to create the executable file we can use to solve specific problems. The way to accomplish that

---

18 Calling a FORTRAN subroutine from a C program can be tricky and often depends on the C and FORTRAN compilers used. In the route here adopted, all arguments to the subroutine must be entered as pointers. Further, the subroutine name must have a suffixed underscore character. Check with your Local Guide for the procedures at your site.

19 See the Local Guide for the meaning of `HEAD` at your site.
objective depends on how things are set up at the local site. In particular, the file defining sgesv.f and files defining all routines supporting sgesv.f must be available in some way. At Lawrence, those files are all compiled and collected into the shared library named sgesvlib.so. Thus, copying that file into the directory containing fem1dla.c and, working in a command window with that directory as its default, we execute the statement

\[
\text{cc -o fdm1dla.xc fdm1dla.c sgesvlib.so}
\]

to create the executable file fdm1dla.xc.

15.8.2 An Example: Simple Harmonic Motion

15.8.2.1 Setting the Problem

We now illustrate the application of fdm1dc to a simplification of Eq. (15.67), specifically the equation

\[
m \frac{d^2 \varphi}{dt^2} + k \varphi = 0 ; \quad 0 \leq t \leq T
\]

for the simple harmonic motion of a mass \( m \) attached to a spring of stiffness \( k \). To match this situation to that discussed in Section 15.1.7, we must interpret \( \alpha \) as a constant equal to \(-m\), \( \beta \) as a constant equal to \( k \), \( L \) as the time \( T \) at the end of the interval of interest, and the independent variable \( x \) as the independent variable \( t \). Further, we must set \( f = 0 \). The dependent variable \( \varphi \) gives the displacement of the oscillator from its equilibrium position. Basically, we are interested in the position over a range of times \( 0 \leq t \leq T \). In a consistent set of units, we will take

\[
m = -\alpha = 4.0 \text{ kg} \quad ; \quad k = \beta = 3.0 \text{ N/m}
\]

To complete the definition of the problem, we need to specify the boundary conditions. Suppose we seek a solution for which \( \varphi(0) = 0 \text{ m} \) and \( \frac{d\varphi(t)}{dt} \big|_{t=T} = 1.0 \text{ m/s} \), i.e., we specify the position at \( t = 0 \text{ s} \) and the velocity at \( t = T \). To effect these conditions, we need to assign the values

\[
p = 0.0 \text{ m} \quad ; \quad \gamma = 0.0 \text{ kg/s} \quad ; \quad q = -4.0 \text{ kg m/s}
\]

the first of which reduces Eq. (15.68) to \( \varphi(0) = 0 \text{ m} \) and the rest of which (with \( \alpha = -4.0 \text{ kg} \)) reduce Eq. (15.69) to \( \frac{d\varphi(t)}{dt} \big|_{t=T} = 1.0 \text{ m/s} \). In this example, we will seek a solution over the interval \( 0 \text{ s} \leq t \leq 10 \text{ s} \), so \( L \rightarrow T = 10.0 \text{ s} \). In physical terms, we start the oscillator at its equilibrium position with an unspecified initial velocity such that, at \( t = 10 \text{ s} \), its velocity will be \( 1.0 \text{ m/s} \).

15.8.2.2 Running the C Program

With these choices, we are now ready to execute the program and solve the problem. If the executable file is in the default directory, the statement

\[
./fdm1dla.xc
\]

submitted to the UNIX operating system starts the program, which begins by requesting input, to which we respond with the values

---

20 Your Local Guide will describe the procedures at your site.
21 Note that we are using a method that requires one item of information at each end of the region of interest—a boundary value problem. More often in problems in motion, one has an initial value problem in which one specifies two items at one end of the region of interest, say an initial position and an initial velocity.
22 See your Local Guide for the procedures at your site.
15.8. USING C TO SOLVE 1D PDES VIA AN FDM

Figure 15.7: Harmonic motion via finite difference analysis. This graph was produced with IDL from data generated by a C program.

Once we have entered these parameters, the program will construct the matrix and generate the solution, ultimately writing it out to the file `fdm1dla_c.dat`.

15.8.2.3 Displaying the Solution Graphically with IDL

To plot the solution using IDL, we would import the data in the file `fdm1d_c.dat` into IDL and invoke the `plot` command with the statements

```
IDL> openr, 1, 'fdm1dla_c.dat'
IDL> readf, 1, n
IDL> soln = fltarr( 2, n )
IDL> readf, 1, soln
IDL> close, 1
IDL> x = soln[0,*] & phi = soln[1,*]
IDL> plot, x, phi, thick = 3, title='N=20', ticklen=1.0
```

The resulting graph is shown in the upper left panel of Fig. 15.7. The remaining frames in this graph were produced by running the program again with $N = 50$ and $N = 100$ and all other parameters the same. The resulting solutions are then plotted in the remaining three frames. The lower right
CHAPTER 15. SOLVING PARTIAL DIFFERENTIAL EQUATIONS

Figure 15.8: Harmonic motion via finite difference analysis. This graph was produced with MATLAB from data generated with a C program.

Frame reveals that, to the resolution of the graph, the solution for $N = 100$ is only slightly different from the solution for $N = 20$, providing evidence of the accuracy of the solution.

15.8.2.4 Displaying the Solution Graphically with MATLAB

To plot the solution using MATLAB, we would import the data in the file `fdm1d_c.dat` into MATLAB and invoke the `plot` command with the statements:

```matlab
id = fopen('fdm1dla_c.dat', 'r');
n = fscanf(id, '%d', 1);
soln = transpose(fscanf(id, '%f', [2,n]));
status = fclose(id);
x = soln(:,1); phi = soln(:,2);
plot(x, phi, 'Color', 'black', 'LineWidth', 4)
title('N=20', 'FontSize', 20)
grid on
```

The resulting graph is shown in the upper left panel of Fig. 15.8. The remaining frames in this graph were produced by running the program again with $N = 50$ and $N = 100$ and all other parameters the same. The resulting solutions are then plotted in the remaining three frames. The lower right frame reveals that, to the resolution of the graph, the solution for $N = 100$ is only slightly different from the solution for $N = 20$, providing evidence of the accuracy of the solution.

15.8.2.5 Displaying the Solution Graphically with OCTAVE

To plot the solution using OCTAVE, we would import the data in the file `fdm1d_c.dat` into OCTAVE and invoke the `plot` command with the statements:

```octave
id = fopen('fdm1dla_c.dat', 'r');
n = fscanf(id, '%d', 1);
soln = transpose(fscanf(id, '%f', [2,n]));
status = fclose(id);
x = soln(:,1); phi = soln(:,2);
plot(x, phi, 'Color', 'black', 'LineWidth', 4)
title('N=20', 'FontSize', 20)
grid on
```
15.8. USING C TO SOLVE 1D PDES VIA AN FDM

Figure 15.9: Harmonic motion via finite difference analysis. This graph was produced with OCTAVE from data generated by a C program.

```c
id = fopen( 'fdm1dla_c.dat', 'r' );
n = fscanf( id, '%d', 1 );
soln = transpose( fscanf( id, '%f', [2,n] ) );
status = fclose( id );
x = soln(:,1); phi = soln(:,2);
plot( x, phi, 'Color', 'black', 'LineWidth', 4 )
title( 'N=20', 'FontSize', 20 )
grid on
```

The resulting graph is shown in the upper left panel of Fig. 15.9. The remaining frames in this graph were produced by running the program again with $N = 50$ and $N = 100$ and all other parameters the same. The resulting solutions are then plotted in the remaining three frames. The lower right frame reveals that, to the resolution of the graph, the solution for $N = 100$ is only slightly different from the solution for $N = 20$, providing evidence of the accuracy of the solution.

15.8.2.6 Displaying the Solution Graphically with PYTHON

To plot the solution using PYTHON, we would import the data in the file `fdm1d.c.dat` into PYTHON and invoke the `plot` command from the `matplotlib.pyplot` module with the statements

```python
import numpy as np
import matplotlib.pyplot as plt
f = open( 'fdm1dla_c.dat', 'r' )
n=int( f.readline() )
soln = []
for line in f:
    soln.append([float(x) for x in line.split()] )
```
Figure 15.10: Harmonic motion via finite difference analysis. This graph was produced with PYTHON from data generated by a C program.

```
f.close()
soln=np.array(soln)
x = soln[:,0]; phi = soln[:,1]
plt.plot( x, phi, color='black', linewidth=3 )
plt.title( 'N=20', fontsize=12 )
plt.grid(color='black')
```

The resulting graph is shown in the upper left panel of Fig. 15.10. The remaining frames in this graph were produced by running the program again with $N = 50$ and $N = 100$ and all other parameters the same. The resulting solutions are then plotted in the remaining three frames. The lower right frame reveals that, to the resolution of the graph, the solution for $N = 100$ is only slightly different from the solution for $N = 20$, providing evidence of the accuracy of the solution.

### 15.9 Finite Element Methods (FEMs) in One Dimension

Finite element methods provide an alternative to finite difference methods for approximating the solution of a boundary value problem—ordinary or partial differential equation plus boundary conditions—in one or more dimensions. The total process involves

1. **Discretizing the domain (preprocessing).** The region of interest is subdivided into a number of small elements by appropriately chosen nodes, at each of which an approximation to the dependent variable $\phi$ will be sought. The discretization of the domain through the identification of suitable nodes and elements is known as preprocessing.

2. **Selecting the interpolation or shape functions.** The interpolation or shape functions for approximating the dependent variable within an element are selected. Because of its simplicity, linear interpolation is commonly used. Higher-order polynomials are more accurate but they also result in a more complicated formulation.
3. **Formulating the equations for a single element.** Equations for each element—the elemental equations—are formulated using the Ritz variational method, the Galerkin method, or some alternative and less common method.

4. **Assembling the system of equations.** The full system of equations is then obtained by (1) assembling the elemental equations into a set applying to the entire region and (2) imposing continuity conditions at the interfaces between elements.

5. **Incorporating the boundary conditions.** Constraints imposed at the boundaries of the region of interest are brought to bear to resolve unknowns left undetermined by the previous steps.

6. **Solving the system of equations.** The (probably large) system of simultaneous, linear, algebraic equations deduced at the previous step is solved for the dependent variable at each node. Such numerical methods as Gauss-Jordan elimination, Gaussian elimination with backsubstitution, and LU (lower-upper) decomposition can all be invoked.

7. **Displaying the solution (postprocessing).** The resulting solution is displayed in various ways. Separate from the finite element method, this last step is known as postprocessing. Postprocessing can be a totally separate process from the other steps, and may even be carried out with different software packages than were used to find the solution itself.

In this and the next section(s), we illustrate the method of finite element analysis by applying these steps to the one-dimensional boundary value problem defined in Section 15.1.7.

### 15.9.1 Discretizing the Domain: Preprocessing

The first step in the finite element method is to divide the region of interest, \( 0 \leq x \leq L \), into a selected number of elements. In the one-dimensional case, these elements will be line segments whose endpoints are called nodes. Let \( M \) be the number of elements, \( N = M + 1 \) be the number of nodes, and \( l^{(e)} \) \((e = 1, 2, 3, \ldots, M)\) be the length of the \( e \)-th element. Furthermore, let \( x_i \) \((i = 1, 2, 3, \ldots, N)\) be the coordinate of the \( i \)-th node. In particular, \( x_1 = 0 \) and \( x_N = L \). The indices \( i \) are known as the global node numbers. In addition, we introduce a local numbering system, in which the nodes of the \( e \)-th element are denoted by \( x_1^{(e)} \) and \( x_2^{(e)} \), with \( x_1^{(e)} < x_2^{(e)} \). In the present one-dimensional context, the locally and globally numbered coordinates of the nodes are related by

\[
\begin{align*}
   x_1^{(e)} &= x_e, & x_2^{(e)} &= x_{e+1} & ; & e = 1, 2, 3, \ldots, M
\end{align*}
\]  

(15.84)

Here, on the left-hand sides, the superscript \( e \) refers to the element and the subscript (1 or 2) refers to the node’s local number. The quantities on the right-hand sides are the globally labeled nodes, and the subscript is a global node number. For example, the two nodes of element 2 are identified locally as \( x_1^{(2)} \) and \( x_2^{(2)} \) and globally as \( x_2 \) and \( x_3 \). The numbering both of elements and of nodes is illustrated in Fig. 15.11, and the relationship between global and local node numbers is conveyed explicitly in the connectivity matrix shown in Table 15.1. Further, with this notation, \( l^{(e)} = x_{e+1} - x_e = x_2^{(e)} - x_1^{(e)} \) and, in general, will vary from element to element.

### 15.9.2 Selecting Interpolation or Shape Functions

The next step is to select interpolation functions or shape functions that can be used to approximate \( \varphi(x) \) within an element. For simplicity, we employ linear functions.\(^{24}\) Thus, in the \( e \)-th element, we approximate the unknown function by

\[
\varphi^{(e)}(x) = a^{(e)} + b^{(e)} x
\]  

(15.85)

\(^{23}\)For variety (and to develop the ability to think in both numbering schemes), we here elect to start the numbering of nodes at \( i = 1 \) rather than at \( i = 0 \).

\(^{24}\)As noted earlier, higher order polynomials may be used as well, though more nodes would then be necessary. See Exercises 15.10, 15.11, and 15.12.
Figure 15.11: Discretization of the solution domain in one dimension: (a) element and global node numbers; (b) element with local node numbers. As illustrated, the spacing of the nodes—and hence the lengths of the elements—need not be uniform.

Table 15.1: The connectivity matrix for the example in Section 15.9.1.

<table>
<thead>
<tr>
<th>Element</th>
<th>Global number of local node 1</th>
<th>Global number of local node 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>...</td>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>M - 1</td>
<td>N - 2</td>
<td>N - 1</td>
</tr>
<tr>
<td>M</td>
<td>N - 1</td>
<td>N</td>
</tr>
</tbody>
</table>

where $a^{(e)}$ and $b^{(e)}$ are constants to be determined. For subsequent convenience, however, it is preferable to express this linear relationship in terms of the values $\tilde{\varphi}_1^{(e)}$ and $\tilde{\varphi}_2^{(e)}$ at the end points of the element rather than in terms of the slope $b^{(e)}$ and intercept $a^{(e)}$. Since these end points occur locally at $x_1^{(e)}$ and $x_2^{(e)}$, we have that

$$
\tilde{\varphi}_1^{(e)} = a^{(e)} + b^{(e)} x_1^{(e)} \\
\tilde{\varphi}_2^{(e)} = a^{(e)} + b^{(e)} x_2^{(e)}
$$

(15.86)

If we solve Eq. (15.86) for $a^{(e)}$ and $b^{(e)}$, substitute the results into Eq. (15.85), and group the terms appropriately, we discover that the approximation in Eq. (15.85) can be written—see Exercise 15.26—in the form

$$
\tilde{\varphi}^{(e)}(x) = \sum_{j=1}^{2} N_j^{(e)}(x) \tilde{\varphi}_j^{(e)}
$$

(15.87)

where

$$
N_1^{(e)}(x) = \frac{x_2^{(e)} - x}{l^{(e)}}
$$

(15.88)

$$
N_2^{(e)}(x) = \frac{x - x_1^{(e)}}{l^{(e)}}
$$

(15.89)
Figure 15.12: The shape functions for linear interpolation.

and \( l^{(e)} = x_2^{(e)} - x_1^{(e)} \) is the length of the element. Equations (15.88) and (15.89) define the shape functions for the \( e \)-th element. Graphs of these shape functions are shown in Fig. 15.12. Note that

\[
\begin{align*}
N_1^{(e)}(x_i^{(e)}) &= 1 & N_1^{(e)}(x_2^{(e)}) &= 0 \\
N_2^{(e)}(x_1^{(e)}) &= 0 & N_2^{(e)}(x_2^{(e)}) &= 1
\end{align*}
\]  

(15.90)

i.e., that one of the shape functions has the value one at the lower end of the element and diminishes linearly to zero at the upper end while the other has the value zero at the lower end and rises linearly to one at the upper end. These properties assure that the sum in Eq. (15.87) has the proper value at each end of the element to which it applies.

### 15.9.3 Formulating the Equations for a Single Element

The next step in the finite element method is to formulate the equations constraining the solution at the nodes defining a single element. We first define the residual \( r(x) \) of Eq. (15.67) as the difference between the right-hand side and the left-hand side when the approximate solution \( \tilde{\varphi} \) is substituted into the equation, i.e., by

\[
r(x) = -\frac{d}{dx} \left( \alpha \frac{d\tilde{\varphi}}{dx} \right) + \beta \tilde{\varphi} - f
\]

(15.91)

Were \( \tilde{\varphi}(x) \) an exact solution, the residual \( r(x) \) would be identically zero. Since \( \tilde{\varphi} \) is only an approximation to \( \varphi \), however, \( r(x) \) will be nonzero. We define the approximate solution by requiring that \( r(x) \) be as small as possible in some sense. We might, for example, choose to make \( r(x) = 0 \) at a discrete set of points. Still better, we can choose to make an appropriate number of weighted “averages” of \( r(x) \) equal to zero, i.e., we can choose \( \varphi_1^{(e)} \) and \( \varphi_2^{(e)} \) so that

\[
R_i^{(e)} = \int_{x_1^{(e)}}^{x_2^{(e)}} W_i^{(e)}(x) r(x) \, dx = 0 \quad ; \quad i = 1, 2
\]

(15.92)

where the \( R_i^{(e)} \) are the weighted residual integrals and the \( W_i^{(e)}(x) \) are as yet unspecified weighting functions for the \( e \)-th element. Note that we need as many weighting functions as we have nodes, since there are that many values of \( \varphi_i^{(e)} \) to determine. Different choices for these weighting functions yield different—though in the end essentially equivalent—methods of solution. In Galerkin’s method, the weighting functions are chosen as the shape functions \( N_i^{(e)}(x) \) used for the expansion of \( \tilde{\varphi} \) in
Eq. (15.87). With that choice, the weighted residual integral for the \( e \)-th element is given by

\[
R_i^{(e)} = \int_{x_1^{(e)}}^{x_2^{(e)}} N_i^{(e)} \left[ - \frac{d}{dx} \left( \alpha(x) \frac{d\phi_i^{(e)}}{dx} \right) + \beta(x) \phi_i^{(e)} \right] dx - \int_{x_1^{(e)}}^{x_2^{(e)}} N_i^{(e)} f(x) dx \quad i = 1, 2 \tag{15.93}
\]

where Eq. (15.91) was substituted into Eq. (15.92) for \( r(x) \). If the first term is integrated by parts, we find that

\[
R_i^{(e)} = \int_{x_1^{(e)}}^{x_2^{(e)}} \left( \alpha(x) \frac{d\phi_i^{(e)}}{dx} \right) dx + \left[ \alpha N_i^{(e)} \frac{d\phi_i^{(e)}}{dx} \right]_{x_1^{(e)}}^{x_2^{(e)}} \quad i = 1, 2 \tag{15.94}
\]

Now if Eq. (15.87) is substituted into Eq. (15.94), we find that

\[
R_i^{(e)} = \sum_{j=1}^{2} \phi_j^{(e)} \int_{x_1^{(e)}}^{x_2^{(e)}} \left( \alpha(x) \frac{dN_i^{(e)}}{dx} \frac{d\phi_j^{(e)}}{dx} + \beta(x) N_i^{(e)} \phi_j^{(e)} \right) dx - \int_{x_1^{(e)}}^{x_2^{(e)}} N_i^{(e)} f(x) dx \quad i = 1, 2 \tag{15.95}
\]

These equations for the weighted residual integrals can also be expressed in matrix form as

\[
\begin{bmatrix}
R_1^{(e)} \\
R_2^{(e)}
\end{bmatrix} =
\begin{bmatrix}
K_{11}^{(e)} & K_{12}^{(e)} \\
K_{21}^{(e)} & K_{22}^{(e)}
\end{bmatrix}
\begin{bmatrix}
\phi_1^{(e)} \\
\phi_2^{(e)}
\end{bmatrix} -
\begin{bmatrix}
b_1^{(e)} \\
b_2^{(e)}
\end{bmatrix} -
\begin{bmatrix}
g_1^{(e)} \\
g_2^{(e)}
\end{bmatrix} \tag{15.96}
\]

or more compactly as

\[
\{ R^{(e)} \} = \{ K^{(e)} \} \{ \phi^{(e)} \} - \{ b^{(e)} \} - \{ g^{(e)} \} \tag{15.97}
\]

where

\[
K_{ij}^{(e)} = \int_{x_1^{(e)}}^{x_2^{(e)}} \left( \alpha(x) \frac{dN_i^{(e)}}{dx} \frac{dN_j^{(e)}}{dx} + \beta(x) N_i^{(e)} N_j^{(e)} \right) dx \tag{15.98}
\]

\[
b_i^{(e)} = \int_{x_1^{(e)}}^{x_2^{(e)}} N_i^{(e)} f(x) dx \tag{15.99}
\]

and

\[
g_i^{(e)} = \left. \left( \alpha N_i^{(e)} \frac{d\phi_i^{(e)}}{dx} \right) \right|_{x_1^{(e)}}^{x_2^{(e)}} = \left. \left( \alpha N_i^{(e)} \frac{d\phi_i^{(e)}}{dx} \right) \right|_{x=x_1^{(e)}} - \left. \left( \alpha N_i^{(e)} \frac{d\phi_i^{(e)}}{dx} \right) \right|_{x=x_2^{(e)}} \tag{15.100}
\]

Note that the matrix \( K \) is necessarily symmetric in \( i \) and \( j \).

In general, the elements will be short, and the functions \( \alpha(x) \) and \( \beta(x) \) will be slowly varying functions of \( x \). Over the range of integration appearing in Eq. (15.98), these functions can often be treated as constants, though with different values for each element. Under those circumstances,

---

\(^{25}\) Alternatively, we could use the Ritz method to formulate the system of equations. The Ritz method is a variational method in which the boundary value problem is formulated in terms of a variational expression or functional. The minimum of this functional corresponds to the governing differential equation under the given boundary conditions. To obtain the approximate solution, the functional is minimized with respect to its variables. In this approach, the choice of the weighting functions, which seemed quite arbitrary in the Galerkin approach, is embedded naturally in the development. In all cases where a variational formulation exists, the Galerkin and Ritz methods are equivalent. The Galerkin method is, however, applicable even in cases for which a variational formulation cannot be found.

\(^{26}\) Recall that \( \int u \, dv = uv - \int v \, du \) where for this case \( u = N_i^{(e)} \) and \( dv = (d/dx) \left( \alpha^{(e)} \frac{d\phi_i^{(e)}}{dx} \right) \) dx.
15.9. FINITE ELEMENT METHODS (FEMS) IN ONE DIMENSION

Eq. (15.98) can be evaluated analytically by taking the derivatives of the shape functions, substituting those derivatives and the shape functions themselves into Eq. (15.98), and integrating over the element. The result is a $2 \times 2$ matrix whose elements are\(^{27}\)

\[
K_{11}^{(e)} = K_{22}^{(e)} = \alpha^{(e)} \frac{l^{(e)}}{6} + \beta^{(e)} \frac{l^{(e)}}{3} \\
K_{12}^{(e)} = K_{21}^{(e)} = -\alpha^{(e)} \frac{l^{(e)}}{6} + \beta^{(e)} \frac{l^{(e)}}{3}
\]

(15.101)

(15.102)

where, in these equations, $\alpha^{(e)}$ and $\beta^{(e)}$ stand for approximate (constant) values of $\alpha(x)$ and $\beta(x)$ appropriate to the $e$-th element, and $l^{(e)} = x^{(e)}_2 - x^{(e)}_1$ is the length of the $e$-th element. Similarly, if $f^{(e)}$ is the (approximate) constant value of $f$ within the $e$-th element, Eq. (15.99) can be evaluated to give

\[
b_1^{(e)} = b_2^{(e)} = f^{(e)} \frac{l^{(e)}}{2}
\]

(15.103)

If, finally, we replace $N_1^{(e)}$ and $N_2^{(e)}$ in Eq. (15.100) with the shape functions in Eqs. (15.88) and (15.89), we have that

\[
g_1^{(e)} = -\alpha(x_1^{(e)}) \frac{d\tilde{\phi}^{(e)}}{dx} \bigg|_{x_1^{(e)}} \\
g_2^{(e)} = \alpha(x_2^{(e)}) \frac{d\tilde{\phi}^{(e)}}{dx} \bigg|_{x_2^{(e)}}
\]

(15.104)

With $[K]$, $\{b\}$, and $\{g\}$ given by Eqs. (15.101)–(15.104), the elemental equation obtained by requiring $R_i^{(e)}$ as given by Eq. (15.96) to be zero for all $i$ is

\[
\begin{bmatrix}
K_{11}^{(e)} & K_{12}^{(e)} \\
K_{21}^{(e)} & K_{22}^{(e)}
\end{bmatrix}
\begin{bmatrix}
\tilde{\phi}_1^{(e)} \\
\tilde{\phi}_2^{(e)}
\end{bmatrix}
= \begin{bmatrix}
\tilde{b}_1^{(e)} \\
\tilde{b}_2^{(e)}
\end{bmatrix}
+ \begin{bmatrix}
g_1^{(e)} \\
g_2^{(e)}
\end{bmatrix}
\]

(15.105)

15.9.4 Assembling the System of Equations

The next step is to assemble the elemental equations for each element into a single (large) set describing all elements. The process of assembly is best illustrated with an example. Suppose we divide the interval $0 \leq x \leq L$ into $M = 3$ elements with $N = M + 1 = 4$ nodes. The elemental equation for the first element is given by Eq. (15.105) with the superscript $(e)$ set to one, i.e., by

\[
\begin{bmatrix}
K_{11}^{(1)} & K_{12}^{(1)} \\
K_{21}^{(1)} & K_{22}^{(1)}
\end{bmatrix}
\begin{bmatrix}
\tilde{\phi}_1^{(1)} \\
\tilde{\phi}_2^{(1)}
\end{bmatrix}
= \begin{bmatrix}
\tilde{b}_1^{(1)} \\
\tilde{b}_2^{(1)}
\end{bmatrix}
+ \begin{bmatrix}
g_1^{(1)} \\
g_2^{(1)}
\end{bmatrix}
\]

(15.106)

which is equivalent to the two equations

\[
K_{11}^{(1)} \tilde{\phi}_1^{(1)} + K_{12}^{(1)} \tilde{\phi}_2^{(1)} = \tilde{b}_1^{(1)} + g_1^{(1)} \\
K_{21}^{(1)} \tilde{\phi}_1^{(1)} + K_{22}^{(1)} \tilde{\phi}_2^{(1)} = \tilde{b}_2^{(1)} + g_2^{(1)}
\]

(15.107)

Note that in Eqs (15.106) and (15.107) we have employed the local numbering system. For the other two elements—also with local node numbers—we have similarly that

\[
K_{11}^{(2)} \tilde{\phi}_1^{(2)} + K_{12}^{(2)} \tilde{\phi}_2^{(2)} = \tilde{b}_1^{(2)} + g_1^{(2)} \\
K_{21}^{(2)} \tilde{\phi}_1^{(2)} + K_{22}^{(2)} \tilde{\phi}_2^{(2)} = \tilde{b}_2^{(2)} + g_2^{(2)}
\]

(15.108)

\(^{27}\)The integrals that appear here can, of course, be evaluated manually. An alternative evaluation using a symbol manipulating program is laid out in Appendix 15.B.
the fourth and fifth equations with their sum to find that Eq. (15.110) becomes

\[ K^{(3)}_{11} \tilde{\varphi}^{(3)} + K^{(3)}_{12} \tilde{\varphi}^{(3)} = b^{(3)} + g^{(3)} \]  

(15.109)

From the relationship between the local and global numbering systems as defined in Eq. (15.84), however, it is clear that \( x_x^{(1)} = x_x^{(2)} = x_2 \) and \( x_x^{(2)} = x_x^{(3)} = x_3 \). Imposing the condition of continuity at nodes 2 and 3, we conclude then that \( \tilde{\varphi}_1^{(1)} = \tilde{\varphi}_1^{(2)} \) and \( \tilde{\varphi}_2^{(2)} = \tilde{\varphi}_3^{(3)} \). Now if we once again invoke the relationship in Eq. (15.84) and recognize the correspondences

\[ \tilde{\varphi}_1^{(4)} \rightarrow \tilde{\varphi}_1, \quad \tilde{\varphi}_2^{(1)} \rightarrow \tilde{\varphi}_2, \quad \tilde{\varphi}_2^{(2)} = \tilde{\varphi}_3 \rightarrow \tilde{\varphi}_3, \quad \text{and} \quad \tilde{\varphi}_3^{(3)} \rightarrow \tilde{\varphi}_4 \]

linking the (approximate) values at locally numbered nodes to the (approximate) values \( \tilde{\varphi}_1, \tilde{\varphi}_2, \tilde{\varphi}_3 \), and \( \tilde{\varphi}_4 \) at the globally numbered nodes, Eqs. (15.107)–(15.109) become

\[ K^{(1)}_{11} \tilde{\varphi}_1 + K^{(1)}_{12} \tilde{\varphi}_2 = b^{(1)} + g^{(1)} \]

(15.110)

\[ K^{(2)}_{11} \tilde{\varphi}_1 + K^{(2)}_{12} \tilde{\varphi}_2 = b^{(2)} + g^{(2)} \]

(15.111)

Thus, we have six equations but only four unknowns. Some of these equations must be redundant. If we choose carefully, we might ignore two of them. Alternatively, we can reduce the number of equations to four by replacing the second and third equations with their sum and, similarly, replacing the fourth and fifth equations with their sum to find that Eq. (15.110) becomes

\[
\begin{bmatrix}
K^{(1)}_{11} & K^{(1)}_{12} & 0 & 0 \\
K^{(2)}_{21} & K^{(2)}_{22} + K^{(2)}_{11} & K^{(3)}_{12} & 0 \\
0 & K^{(3)}_{21} & K^{(3)}_{22} + K^{(3)}_{11} & K^{(3)}_{12} \\
0 & 0 & K^{(3)}_{21} & K^{(3)}_{22}
\end{bmatrix}
\begin{bmatrix}
\tilde{\varphi}_1 \\
\tilde{\varphi}_2 \\
\tilde{\varphi}_3 \\
\tilde{\varphi}_4
\end{bmatrix}
= 
\begin{bmatrix}
b^{(1)}_1 \\
b^{(2)}_2 + b^{(2)}_1 \\
b^{(3)}_2 + b^{(3)}_1 \\
g^{(3)}_2 + g^{(3)}_1
\end{bmatrix}
+ 
\begin{bmatrix}
g^{(1)}_1 \\
g^{(2)}_2 + g^{(2)}_1 \\
g^{(3)}_2 + g^{(3)}_1
\end{bmatrix}
\]

(15.112)

or more compactly

\[
[K] \{\tilde{\varphi}\} = \{b\} + \{g\}.
\]

(15.113)

In Eq. (15.112), \([K]\) is the assembled stiffness matrix for the three-element problem. The extension to more than three elements is now evident. Note that, no matter how many elements we have in a one-dimensional problem divided into elements with two nodes, the matrix \([K]\) will always be tridiagonal.

We complete this step in the process by working out the elements in the assembled equation. From Eqs. (15.101), (15.102), and (15.111), we conclude that the nonzero elements of \([K]\) can be written as

\[
K_{11} = K^{(1)}_{11} = \frac{\alpha^{(1)}_l l^{(1)} + \beta^{(1)} l^{(1)}}{3}
\]

\[
K_{ii} = K^{(i-1)}_{22} + K^{(i)}_{11}
= \frac{\alpha^{(i-1)} l^{(i-1)} + \beta^{(i-1)} l^{(i-1)}}{3} + \frac{\alpha^{(i)} l^{(i)} + \beta^{(i)} l^{(i)}}{3} \quad ; \quad i = 2, 3, 4, \ldots, N - 1
\]
28
proximate) values at the nodes—now assumes the form
the right-hand side of Eq. (15.116) is zero for

\[ K_{i+1,i} = K_{i,i+1} = K_{12} = \frac{\alpha^{(i)} + \beta^{(i)} \ell^{(i)}_2}{6} ; \quad i = 1, 2, 3, \ldots, N - 1 \tag{15.113} \]

where (by way of reminder) \( N = M + 1 \) is the number of nodes and \( M \) is the number of elements. Similarly, using Eqs. (15.103) and (15.111), we can write the elements of \( \{ b \} \) in the form

\[ b_1 = b_1^{(1)} = f^{(1)} \frac{\ell^{(1)}_2}{2} \]
\[ b_i = b_2^{(i-1)} + b_1^{(i)} = f^{(i-1)} \frac{\ell^{(i-1)}_2}{2} + f^{(i)} \frac{\ell^{(i)}_2}{2} \quad i = 2, 3, 4, \ldots, N - 1 \]
\[ b_N = b_1^{(M)} = f^{(M)} \frac{\ell^{(M)}_2}{2} \tag{15.114} \]

Finally, we need to evaluate the elements of \( \{ g \} \). Note that all but the first and last entries in \( \{ g \} \) can be written as

\[ g_i = g_2^{(i-1)} + g_1^{(i)} \tag{15.115} \]

If Eq. (15.104) is substituted into Eq. (15.115), we have

\[ g_i = \alpha \left. \frac{d\tilde{\varphi}}{dx} \right|_{x = x_2^{(i-1)}} - \alpha \left. \frac{d\tilde{\varphi}}{dx} \right|_{x = x_1^{(i)}} \tag{15.116} \]

However, \( x_2^{(i-1)} = x_1^{(i)} = x_i \) [see Eq. (15.84)] and \( \alpha (d\tilde{\varphi}/dx) \) must be continuous at \( x_i \). As a result, the right-hand side of Eq. (15.116) is zero for \( i = 2, 3 \) and we are left with the vector

\[ \{ g \} = \begin{bmatrix} -\alpha(x_1) \left. \frac{d\tilde{\varphi}}{dx} \right|_{x = x_1} \\ 0 \\ 0 \\ \alpha(x_4) \left. \frac{d\tilde{\varphi}}{dx} \right|_{x = x_4} \end{bmatrix} \tag{15.117} \]

In general for a problem with \( M \) elements (\( N \) nodes), we would find that

\[ g_1 = -\alpha(x_1) \left. \frac{d\tilde{\varphi}}{dx} \right|_{x = x_{1} = 0} \]
\[ g_i = 0 \quad i = 2, 3, 4, \ldots, N - 1 \]
\[ g_N = \alpha(x_N) \left. \frac{d\tilde{\varphi}}{dx} \right|_{x = x_N = L} \tag{15.118} \]

In consequence of the considerations to this point, our system of equations for \( \{ \tilde{\varphi} \} \)—the (approximate) values at the nodes—now assumes the form\(^{28}\)

\[ \begin{bmatrix} \begin{bmatrix} K_{11} & K_{12} & K_{13} & K_{14} \\ K_{21} & K_{22} & K_{23} & K_{24} \\ K_{31} & K_{32} & K_{33} & K_{34} \\ K_{41} & K_{42} & K_{43} & K_{44} \end{bmatrix} \end{bmatrix} \begin{bmatrix} \tilde{\varphi}_1 \\ \tilde{\varphi}_2 \\ \tilde{\varphi}_3 \\ \tilde{\varphi}_4 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} + \begin{bmatrix} g_1 \\ 0 \\ 0 \\ g_4 \end{bmatrix} \tag{15.119} \]

Similar expressions can be readily written for cases in which there are more nodes than four.

\(^{28}\)We elect to introduce a symbol for every element in \( K \) even though many of those elements are known in advance to be zero.
15.9.5 Incorporating the Boundary Conditions

Before we can solve Eq. (15.119) for \( \{ \tilde{\varphi} \} \), we need to incorporate the boundary conditions. First consider the boundary condition of the third kind given in Eq. (15.69). If we replace \( \varphi \) by its approximation \( \tilde{\varphi} \), solve for \( \alpha (d\tilde{\varphi}/dx) \) at \( x = L \), and substitute this into Eq. (15.118), we have that

\[
g_N = \alpha \left. \frac{d\tilde{\varphi}}{dx} \right|_{x=L} = q - \gamma \tilde{\varphi}_N
\]  

(15.120)

or specifically for our case that

\[
g_4 = \alpha \left. \frac{d\tilde{\varphi}}{dx} \right|_{x=L} = q - \gamma \tilde{\varphi}_4
\]  

(15.121)

Thus, the last equation in our system becomes

\[
K_{41} \tilde{\varphi}_1 + K_{42} \tilde{\varphi}_2 + K_{43} \tilde{\varphi}_3 + K_{44} \tilde{\varphi}_4 = b_4 + q - \gamma \tilde{\varphi}_4
\]  

(15.122)

which we can recast as

\[
K_{41} \tilde{\varphi}_1 + K_{42} \tilde{\varphi}_2 + K_{43} \tilde{\varphi}_3 + (K_{44} + \gamma) \tilde{\varphi}_4 = b_4 + q
\]  

(15.123)

With this rewriting of the fourth equation, our system of equations becomes

\[
\begin{bmatrix}
K_{11} & K_{12} & K_{13} & K_{14} \\
K_{21} & K_{22} & K_{23} & K_{24} \\
K_{31} & K_{32} & K_{33} & K_{34} \\
K_{41} & K_{42} & K_{43} & K_{44} + \gamma
\end{bmatrix}
\begin{bmatrix}
\tilde{\varphi}_1 \\
\tilde{\varphi}_2 \\
\tilde{\varphi}_3 \\
\tilde{\varphi}_4
\end{bmatrix}
= \begin{bmatrix}
b_1 \\
b_2 \\
b_3 \\
b_4 + q
\end{bmatrix}
\]  

(15.124)

In general for a domain with \( N \) nodes, we have

\[
K_{NN} \rightarrow K_{NN} + \gamma = \frac{a^{(M)}}{l^{(M)}} + \frac{\beta^{(M)}}{3} + \frac{\gamma}{2}
\]  

(15.125)

and hence the \( N \)th element of \( \{ g \} \) is absorbed in \( [K] \) and \( \{ b \} \). This operation can always be performed for mixed (and Neumann) boundary conditions.

Imposing the Dirichlet boundary condition of Eq. (15.68) is simpler. Recall that for \( M \) elements, we have \( M+1 \) unknowns \( \tilde{\varphi}_i \) \( (i = 1, 2, 3, \ldots, N) \) and \( M+1 \) equations. However, the Dirichlet boundary condition given in Eq. (15.68) specifies the value of one of these unknowns, specifically \( \tilde{\varphi}_1 \). Thus we actually have \( M + 2 \) equations that need to be simultaneously satisfied. But since we only need as many equations as unknowns, we can replace the first equation of Eq. (15.124) with Eq. (15.68). As a result, \( g_1 \) no longer plays a role in the system of equations, and the entire \( g \) vector has been absorbed.\(^{29}\) The new system of equations for our three-element example is therefore

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
K_{21} & K_{22} & K_{23} & K_{24} \\
K_{31} & K_{32} & K_{33} & K_{34} \\
K_{41} & K_{42} & K_{43} & K_{44}
\end{bmatrix}
\begin{bmatrix}
\tilde{\varphi}_1 \\
\tilde{\varphi}_2 \\
\tilde{\varphi}_3 \\
\tilde{\varphi}_4
\end{bmatrix}
= \begin{bmatrix}
p \\
b_2 \\
b_3 \\
b_4
\end{bmatrix}
\]  

(15.127)

where \( K_{44} \) and \( b_4 \) have been modified according to Eqs. (15.125) and (15.126).

\(^{29}\) Actually the original first equation \( K_{11} \tilde{\varphi}_1 + K_{12} \tilde{\varphi}_2 + K_{13} \tilde{\varphi}_3 + K_{14} \tilde{\varphi}_4 = b_1 + g_1 \) is merely reinterpreted as an equation determining \( g_1 \) after the solution \( \{ \tilde{\varphi} \} \) has been found.
The symmetry of the coefficient matrix is easily restored. Consider, for example, the second equation in Eq. (15.127), namely

\[ K_{21} \tilde{\varphi}_1 + K_{22} \tilde{\varphi}_2 + K_{23} \tilde{\varphi}_3 + K_{24} \tilde{\varphi}_4 = b_2 \]  

(15.128)

Since \( \tilde{\varphi}_1 = p \), however, this equation can be recast as

\[ K_{22} \tilde{\varphi}_2 + K_{23} \tilde{\varphi}_3 + K_{24} \tilde{\varphi}_4 = b_2 - K_{21} p \]  

(15.129)

A similar recasting of the other equations leads finally to

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & K_{22} & K_{23} & K_{24} \\
0 & K_{32} & K_{33} & K_{34} \\
0 & K_{42} & K_{43} & K_{44}
\end{bmatrix} \begin{bmatrix}
\tilde{\varphi}_1 \\
\tilde{\varphi}_2 \\
\tilde{\varphi}_3 \\
\tilde{\varphi}_4
\end{bmatrix} = \begin{bmatrix}
p \\
b_2 - K_{21} p \\
b_3 - K_{31} p \\
b_4 - K_{41} p
\end{bmatrix}
\]  

(15.130)

and once again we have a symmetric system. Further, since all elements in the coefficient matrix and all elements in the vector of inhomogenieties are now known, these four equations uniquely determine the vector \( \{ \tilde{\varphi} \} \) of unknowns.\(^{30}\)

We can carry our analysis no further without selecting a particular programming language in which to implement explicit coding. That task is undertaken in the next several sections.

15.15 Using C to Solve 1D PDEs via an FEM

15.15.1 A General Coding using LAPACK

The final step is to solve Eq. (15.130) for the unknowns \( \tilde{\varphi}_0, \tilde{\varphi}_1, \ldots, \tilde{\varphi}_N \). A program to construct Eq. (15.130) and then solve the resulting system for \( \tilde{\varphi} \) at each node would begin by asking the user to input the values of the various parameters involved in the problem. In all cases, the parameters \( p, \gamma, \) and \( q \), which relate to the boundary conditions, are constants. In general, the quantities \( \alpha, \alpha', \beta, f, \) and \( l(e) \), which relate to the physical system involved, and \( f \), which represents a source or excitation function will depend on \( x \). Further, \( l(e) \), which reflects the particular discretization adopted, will in general depend on the element. For simplicity in illustration, however, we will suppose these five quantities to be constants as well.\(^{31}\) Indeed, if \( \alpha \) is constant, then \( \alpha' \) is zero, and we will simply leave out terms multiplied by \( \alpha' \) when we construct the equations.

The first segment of the program we wish to write will, then, request input of all the constants except\(^{32}\) \( M \) needed in the remainder of the program. Appropriate C statements for all but \( M \) are\(^{33}\)

```c
#include <stdio.h>
#include <math.h>
float alpha, beta, f, l, p, gamma, q;

printf("Enter alpha: "); scanf("%f", &alpha);
printf("Enter beta: "); scanf("%f", &beta);
```

\(^{30}\)We assume, of course, that the coefficient matrix is not singular.

\(^{31}\)If \( \alpha, \alpha', \beta, f, \) and \( l(e) \) are not constant, each would have to be represented by an appropriately sized vector, and values would have to be given for all elements in these vectors.

\(^{32}\)As explained later, we elect to use the LAPACK routine \texttt{sgeev.f} to solve the linear equation to be constructed. That routine requires as input for \( K \) a two-dimensional array that has the right dimensions for the problem. Since dynamic allocation of two-dimensional arrays in C is an advanced topic, we elect to hard code the number of segments in the program so the arrays can be given the proper dimension for the problem.

\(^{33}\)Variable declaration statements will be included when the variables are introduced but will be collected at the beginning of the program in a final listing.
printf("Enter f: "); scanf("%f", &f);
printf("Enter l: "); scanf("%f", &l);
printf("Enter p: "); scanf("%f", &p);
printf("Enter gamma: "); scanf("%f", &gamma);
printf("Enter q: "); scanf("%f", &q);

The next step is to assemble $[K]$ as given in Eqs. (15.111) and (15.113). To that end, the several statements

```c
#define M 20
float K[M+1][M+1];
int i, j;
for( i=0; i<=M; i++)
{  B[i][0] = 0.0; IPIV[i]=0;
   for(j=0; j<=M; j++) K[i][j] = 0.0;
}
```

(1) specify the selected number of elements into which the system will be divided,34 (2) declare the properly-sized variable $K$, (3) declare additional variables, and (4) initialize the variable $K$. Then, using the statements

```c
float s, s2, T;
s = alpha/l + beta*l/3.0; /* Evaluate common quantities */
s2 = 2.0*s;
T = -alpha/l + beta*l/6.0;

K[0][0] = s; /* Set diagonal elements of $K$ */
for(i=1; i<=M; i++) K[i][i]=s2;
K[M][M} = s;

for(i=0;i<=M-1; i++) /* Set elements above and below */
{  K[i+1][i] = T; /* main diagonal of $AK$ */
   K[i][i+1] = T;
}
```

we assign values to the diagonal and non-zero off-diagonal elements in accordance with Eq. (15.113).35,36

To complete the assembly, we create the vector $B$ as given by Eq. (15.114) in a similar fashion, invoking the statements

```c
float B[M+1][1], u, u2;
u = f*l/2.0; u2 = 2.0*u; /* Evaluate common quantities */

B[0][0] = u; /* Set elements of $B$ */
for(i=1,i<M; i++) B[i][0] = u2;
B[M][0] = u;
```

Note that, because the LAPACK routine to be used (sgesv) allows for simultaneous solution for more than one set of inhomogenieties by adding columns to the vector $B$, that vector must be dimensioned as a two-dimensional matrix with, in our case, only one column.

---

34This is the only statement that will need to be changed to solve the problem with a different number of elements.
35Even though the coefficient matrix in this case is tridiagonal, we elect here not to use the LAPACK routine routine sgtav but instead to illustrate an approach that will work even if the coefficient matrix is not tridiagonal.
36Remember that indices in C start at zero, so all C indices are one less than the corresponding subscript on $K$. 
Finally, to produce the system of equations given in Eq. (15.130), we must incorporate the boundary conditions and thus modify $K$ and $B$. First, we impose the boundary condition of the third kind through Eqs. (15.125) and (15.126) with the statements

$$
K[M][M] = K[M][M] = \text{gamma};$
$$
B[M][0] = B[M][0] + q;
$$

Then we impose the Dirichlet condition as described in Section 15.9.5 with the statements

$$
K[0][0] = 1.0;
$$
$$
B[0][0] = p;
$$

for($j=1; j<=M; j++) K[j][0] = 0.0;

for($i=1; i<=M; i++) B[i][0] = B[i][0] - K[0][i]*p;

for($i=1; i<=M; i++) K[0][i] = 0.0;

As in Eq. (15.130), $K_{11}$, or $AK[0][0]$ in C terminology, is set equal to one and $b_1$, or $B[0][0]$, is set equal to $p$. Then, we assign the value zero to $K_{1j}$ for $j = 2, 3, 4, \ldots, M$ ($AK[0,j]$ for $j = 1, 2, 3, 4, \ldots, M$). Finally, the last two lines multiply all but the first entry in the first column of $K$ by $p$, subtract each of those values from the corresponding element of $b$ and then—note the order of operations—set all but the first element in the first column of $K$ to zero. As a result, we now have in $K$ and $B$ the coefficient matrix and the vector of inhomogeneties as in Eq. (15.130), a symmetric system of equations ready to be solved.

For the final solution of this system of equations, we elect to use the LAPACK routine \texttt{sgesv.f}, which is a single-precision (s) routine for solving general\textsuperscript{37} (ge) simultaneous linear algebraic equations.\textsuperscript{38} All of the required inputs to this routine when called from a C program must be provided as pointers. The appropriate C statements are

$$
\text{float IPIV[M+1];}
$$
$$
\text{int MM, INFO;}
$$
$$
i = 1; \text{MM} = M+1;
$$
$$
\text{sgesv')(&MM, &i, &K, &MM, &IPIV, &B, &MM, &INFO);} 
$$

Here,

\begin{itemize}
\item The first argument, MM, on input is the number of equations; it is not changed on output.
\item The second argument, i, on input is the number of columns in B; it is not changed on output.
\item The third argument, K, on input is the coefficient matrix; on output it contains the LU decomposition of the input matrix.
\item The fourth argument, MM, on input is the number of rows in AK; it is not changed on output.
\item The fifth argument, IPIV, on input is not used; on output it is an integer array conveying the exchanges of rows/columns made in the process of working out the solution. It must be dimensioned as a one-dimensional integer array of size equal to the number of equations.
\item The sixth argument B on input is the array of inhomogeneties; on output it is the array of solutions, which is symbolized $\tilde{\phi}$ in Eq. (15.130).
\item The seventh argument, MM, on input is the number of rows in B; it is not changed on output.
\item The eighth argument, INFO, on input is not used; on output its value conveys whether or not the solution is successful.
\end{itemize}

\textsuperscript{37}Systems whose coefficient matrix is not necessarily tridiagonal or symmetric.

More detailed information about these variables is compiled in the comments at the beginning of the LAPACK file *sgesv.f*

At this point, B contains the (approximate) solution to the boundary value problem defined in Section 15.1.7. To make this solution available for a graphical visualization program, we need finally to write its dimension and the values into a file, so we add the statements

```c
FILE *fptr;
fptr = fopen( "fem1dla_c.dat", "w" );
fprintf( fptr, "%d\n",M+1 );
for( i=0; i<=M; i++) fprintf( fptr, "%7.2f %10.4f\n", l*i, B[i][0] );
fclose( fptr );
```

to our program, which is now complete. Note that we have added to the file the calculation of the value of the independent variable corresponding to the solution at each node. Thus, the file will have a first line containing the number of lines in the rest of the file, and each of those lines contains two values, the first being the x coordinate of a node and the second being the solution at that node.

All of the preceding C code has been incorporated into the program *fem2dla.c*, a commented listing of which can be found in Appendix 15.C.6. (The file itself can be copied from the directory `$HEAD/cc`.)

We are at long last ready to compile and link this program with the needed LAPACK routines to create the executable file we can use to solve specific problems. The way to accomplish that objective depends on how things are set up at the local site. In particular, the file defining *sgesv.f* and files defining all routines supporting *sgesv.f* must be available in some way. At Lawrence, those files are all compiled and collected into the shared object library named *sgesvlib.so*. Thus, copying that file into the directory containing *fem1dla.c* and, working in a command window with that directory as its default, we execute the statement

```
cc -o fem1dla.xc fem1dla.c sgesvlib.so -lm
```

to create the executable file *fem1dla.xc*.

### 15.15.2 An Example: Simple Harmonic Motion

#### 15.15.2.1 Setting the Problem

We now illustrate the application of *fem1dla.c* to a simplification of Eq. (15.67), specifically the equation

\[
m \frac{d^2 \varphi}{dt^2} + k \varphi = 0 \quad ; \quad 0 \leq t \leq T
\]  

(15.131)

for the simple harmonic motion of a mass \( m \) attached to a spring of stiffness \( k \). To match this situation to that discussed in Section 15.1.7, we must interpret \( \alpha \) as a constant equal to \(-m\), \( \beta \) as a constant equal to \( k \), \( L \) as the time \( T \) at the end of the interval of interest, and the independent variable \( x \) as the independent variable \( t \). Further, we must set \( f = 0 \). The dependent variable \( \varphi \) gives the displacement of the oscillator from its equilibrium position. Basically, we are interested in the position over a range of times \( 0 \leq t \leq T \). In a consistent set of units, we will take

\[
m = -\alpha = 4.0 \text{ kg} \quad ; \quad k = \beta = 3.0 \text{ N/m}
\]

\[39\text{Your Local Guide will describe the procedures at your site.}\]
To complete the definition of the problem, we need to specify the boundary conditions. Suppose we seek a solution for which $\varphi(0) = 0$ m and $d\varphi(t)/dt|_{t=T} = 1.0$ m/s, i.e., we specify the position at $t = 0$ s and the velocity at $t = T$. To effect these conditions, we need to assign the values

$$p = 0.0 \text{ m} ; \quad \gamma = 0.0 \text{ kg/s} ; \quad q = -4.0 \text{ kg m/s}$$

the first of which reduces Eq. (15.68) to $\varphi(0) = 0$ m and the rest of which (with $\alpha = -4.0$ kg) reduce Eq. (15.69) to $d\varphi(t)/dt|_{t=T} = 1.0$ m/s. In this example, we will seek a solution over the interval $0 \leq t \leq 10$ s, so $L \rightarrow T = 10.0$ s. In physical terms, we start the oscillator at its equilibrium position with an unspecified initial velocity such that, at $t = 10.0$ s, its velocity will be 1.0 m/s.

### 15.15.2.2 Running the C Program

With these choices, we are now ready to run the program to solve the problem. We invoke the statement

```
./fem1dla.xc
```

to start the process. The first several statements will request input, to which we will respond with the values

```
Enter alpha:  -4.0
Enter beta:   3.0
Enter f:      0.0
Enter l:      0.5
Enter p:      0.0
Enter gamma:  0.0
Enter q:      -4.0
```

Once we have entered these parameters, the remaining statements will construct all necessary matrices and vectors and then generate the solution, ultimately writing it to the file `fem1dla_c.dat`. We will have no further interaction with the program and, presently, the prompt from the operating system will return.

### 15.15.2.3 Displaying the Solution Graphically with IDL

To plot this solution using IDL, we would begin by opening and reading the file with the statements

```
IDL> openr, 1, 'fem1dla_c.dat'
IDL> readf, 1, n & n = fix(n)
IDL> soln = fltarr(2,n)
IDL> readf, 1, soln
IDL> close, 1
```

At this point, `soln` is a two-column array in which the first column `soln[0,*]` contains values of the independent variable $x$ and the second column `soln[1,*]` contains values of the dependent variable $\tilde{\varphi}$. We plot this solution with the statement

```
IDL> plot, soln[0,*], soln[1,*], thick=3, title='N=20', ticklen=1.0
```
Figure 15.13: Harmonic motion via finite element analysis. This graph was produced with IDL from a data file generated by a C program.

The resulting graph is shown in the upper left panel of Fig. 15.13. The remaining frames in this graph were produced by running the program again with $M = 50, l = 0.2$ and $M = 100, l = 0.1$ and all other parameters the same. The resulting solutions are then plotted in the remaining three frames. The lower right frame reveals that, to the resolution of the graph, the solution for $M = 100$ is only slightly different from the solution for $M = 20$, providing evidence of the accuracy of the solution.

15.15.2.4 Displaying the Solution Graphically with MATLAB

To plot this solution using MATLAB, we would begin by opening and reading the file with the statements

```matlab
id = fopen( 'fem1dla_c.dat', 'r' );
n = fscanf( id, '%d', 1 );
soln = transpose( fscanf( id, '%f', [2,n] ) );
status = fclose( id );
```

At this point, `soln` is a two-column array in which the first column `soln(:,1)` contains values of the independent variable $x$ and the second column `soln(:,2)` contains values of the dependent variable $\tilde{\phi}$. We plot this solution with the statement

```matlab
plot( soln(:,1), soln(:,2), 'Color', 'black', 'LineWidth', 4 )
title( 'N=20', 'FontSize', 20 )
grid on
```

The resulting graph is shown in the upper left panel of Fig. 15.14. The remaining frames in this

---

40Note that we are using a method that requires one item of information at each end of the region of interest—a boundary value problem. More often in problems in motion, one has an initial value problem in which one specifies
Figure 15.14: Harmonic motion via finite element analysis. This graph was produced with MATLAB from a data file generated by a C program.

graph were produced by running the program again with $M = 50, l = 0.2$ and $M = 100, l = 0.1$ and all other parameters the same. The resulting solutions are then plotted in the remaining three frames. The lower right frame reveals that, to the resolution of the graph, the solution for $M = 100$ is only slightly different from the solution for $M = 20$, providing evidence of the accuracy of the solution.

15.15.2.5 Displaying the Solution Graphically with OCTAVE

To plot this solution using OCTAVE, we would begin by opening and reading the file with the statements

```octave
id = fopen( 'fem1dla_c.dat', 'r' );
n = fscanf( id, '%d', 1 );
soln = transpose( fscanf( id, '%f', [2,n] ) );
status = fclose( id );
```

At this point, `soln` is a two-column array in which the first column `soln(:,1)` contains values of the independent variable $x$ and the second column `soln(:,2)` contains values of the dependent variable $\tilde{\varphi}$. We plot this solution with the statement

```octave
plot( soln(:,1), soln(:,2), 'Color', 'black', 'LineWidth', 4 )
title( 'N=20', 'FontSize', 20 )
grid on
```

The resulting graph is shown in the upper left panel of Fig. 15.15. The remaining frames in this

two items at one end of the region of interest, say an initial position and an initial velocity.

41Remember that we have written the program `fem1dla.c` to expect $l = L/M$. 
Figure 15.15: Harmonic motion via finite element analysis. This graph was produced with OCTAVE from a data file generated by a C program.

graph were produced by running the program again with \( M = 50, l = 0.2 \) and \( M = 100, l = 0.1 \) and all other parameters the same. The resulting solutions are then plotted in the remaining three frames. The lower right frame reveals that, to the resolution of the graph, the solution for \( M = 100 \) is only slightly different from the solution for \( M = 20 \), providing evidence of the accuracy of the solution.

### 15.15.2.6 Displaying the Solution Graphically with PYTHON

To plot the solution using PYTHON, we would begin by importing needed modules opening and reading the file with the statements

```python
import numpy as np
import matplotlib.pyplot as plt
f = open('fem1dla_c.dat', 'r')
n=int( f.readline() )
soln = [] # Create empty list
for line in f:
    soln.append([float(x) for x in line.split()])
f.close()
soln=np.array(soln) # Convert list to array
x = soln[:,0]; phi = soln[:,1]
plt.plot( x, phi, color='black', linewidth=3 )
plt.title( 'N=20', fontsize=12 )
plt.grid(color='black')
```

At this point, `soln` is a two-column array in which the first column `soln[:,0]` contains values of \( x \) and the second column `soln[:,1]` contains the associated values of the solution. Then, we extract \( x \) and \( \phi \) and plot the solution with the statements
Figure 15.16: Harmonic motion via finite element analysis. This graph was produced with PYTHON from a data file generated by a C program.

The resulting graph is shown in the upper left panel of Fig. 15.16. The remaining frames in this graph were produced by running the program again with \( M = 50, l = 0.2 \) and \( M = 100, l = 0.1 \) and all other parameters the same. The resulting solutions are then plotted in the remaining three frames. The lower right frame reveals that, to the resolution of the graph, the solution for \( M = 100 \) is only slightly different from the solution for \( M = 20 \), providing evidence of the accuracy of the solution.

15.16 Finite Difference Methods (FDMs) in Two Dimensions

We can easily extend the one-dimensional FDM already described to apply to problems involving two independent variables. We must, however, treat initial value problems, such as those involving the wave and diffusion equations, differently from boundary value problems, such as those involving the Laplace equation.

15.16.1 The Wave Equation

As laid out in Section 15.1.1, the wave equation for waves in one dimension involves two independent variables, a spatial coordinate \( x \) locating a point in the one-dimensional medium and a time coordinate \( t \). The displacement \( u(x, t) \) of the medium satisfies Eq. (15.13),

\[
\frac{1}{c^2} \frac{\partial^2 u}{\partial t^2} = \frac{\partial^2 u}{\partial x^2} \quad (15.132)
\]

which is to be solved subject both to the initial conditions

\[
u(x, 0) = f(x) \quad ; \quad \frac{\partial u}{\partial t}(x, 0) = g(x) \quad (15.133)
\]
that specify how the motion of the string is initiated and to appropriate boundary conditions that specify how the medium is “fastened” at its two ends, say at \( x = 0 \) and \( x = L \). To illustrate two distinct types of boundary conditions, we suppose here that the displacement of the medium must be zero for all time at \( x = 0 \) and the derivative of that displacement must be zero for all time at \( x = L \), i.e., that

\[
\begin{align*}
  u(0, t) &= 0 \quad ; \\
  \frac{\partial u}{\partial x}(L, t) &= 0
\end{align*}
\]  

(15.134)

For a string, for which \( u(x, t) \) represents a displacement, these conditions stipulate physically that the string is tied down at \( x = 0 \) and free at \( x = L \); for an air column, for which \( u(x, t) \) also represents a displacement, these conditions stipulate that the pipe containing the column is closed at \( x = 0 \) (where there will therefore be a displacement node and a pressure antinode) and open at \( x = L \) (where there will be a displacement antinode and a pressure node).

In one numerical approach to this problem via an FDM, we introduce a set of equally spaced nodes in the spatial dimension but leave the temporal dimension continuous. Suppose we divide the interval \( 0 \leq x \leq L \) into \( N \) elements by \( N + 1 \) nodes, each separated from its nearest neighbor(s) by the distance \( \Delta x = L/N \). We denote the coordinates of those nodes by \( x_i = i \Delta x \) with \( i = 0, 1, 2, \ldots, N \). In particular, \( x_0 = 0 \) and \( x_N = L \). Then, we introduce the set of functions \( u_i(t) = u(x_i, t) \), evaluate the PDE of Eq. (15.132) at \( x_i \), and express the spatial derivative at \( x_i \) in terms of finite differences as in Eq. (15.75) to find that

\[
d_2^2 u_i \frac{dt^2}{dt^2} = \frac{c^2}{\Delta x^2} \left( u_{i+1} - 2u_i + u_{i-1} \right) \)  

(15.135)

The boundary conditions can now be invoked to resolve problems with this equation when \( i = 0 \) and \( i = N \), at which points the equation refers to \( u \) at points outside the domain of the problem, i.e., to \( u_{-1} \) or \( u_{N+1} \), respectively. At \( x = 0 \), the boundary condition implies that \( u_0 = 0 \), so we can replace Eq. (15.135) at \( i = 0 \) with the equation \( u_0 = 0 \), though—for the sake of a similar treatment of all nodes—it is more appropriate to regard \( u_0 = 0 \) as the solution to the initial value problem

\[
\begin{align*}
  d_2^2 u_0 \frac{dt^2}{dt^2} &= 0 \quad ; \\
  u_0(0) &= 0 \quad , \quad \frac{du_0}{dt}(0) = 0
\end{align*}
\]  

(15.136)

To address Eq. (15.135) at \( i = N \), we look to a central difference approximation of the spatial derivative, finding that

\[
\frac{\partial u}{\partial x}(L, t) \approx \frac{u_{N+1}(t) - u_{N-1}(t)}{2\Delta x} = 0 \quad \Rightarrow \quad u_{N+1} = u_{N-1}
\]  

(15.137)

Then, we write Eq. (15.135) for \( i = N \), substitute \( u_{N-1} \) for \( u_{N+1} \), and solve the result for \( d_2^2 u_N \frac{dt^2}{dt^2} \). In this approach, we have discretized only the spatial variable, and the task of solving a PDE in two variables becomes one of solving the coupled set

\[
\begin{align*}
  d_2^2 u_0 \frac{dt^2}{dt^2} &= 0 \\
  d_2^2 u_i \frac{dt^2}{dt^2} &= \frac{c^2}{\Delta x^2} \left( u_{i+1} - 2u_i + u_{i-1} \right) , \quad i = 1, 2, 3, \ldots, N - 1 \\
  d_2^2 u_N \frac{dt^2}{dt^2} &= \frac{c^2}{\Delta x^2} \left( -2u_N + 2u_{N-1} \right)
\end{align*}
\]  

(15.138-15.140)

of ODEs subject to the initial conditions

\[
\begin{align*}
  u_i(0) &= f(x_i) \quad ; \\
  \frac{du_i}{dt}(0) &= g(x_i)
\end{align*}
\]  

(15.141)
Figure 15.17: Geometry for full discretization of the wave equation. Equation 15.143 involves the points marked with solid figures and has been solved to express the solution at the (one) point marked with a solid square in terms of the solutions at the (four) points marked with solid circles.

where, to reflect properly the conditions imposed on $u_0$, we suppose that $f(x_0) = f(0) = 0$ and $g(x_0) = g(0) = 0$. By discretizing the spatial variable, we have reduced our problem to a problem of the sort addressed in Chapter 11, and we therefore say no more about it here.

We could go one step further, discretizing also the time variable by introducing a time step $\Delta t$ and the discrete time instants $t_j = j \Delta t$, $j = 0, 1, 2, \ldots$. Then, approximating both the spatial and the temporal derivatives by finite differences and introducing the quantities $u_{i,j} = u(x_i, t_j)$, we might evaluate the original PDE of Eq. (15.132) at $(x, t) = (x_i, t_j)$ and find the fully discretized form

$$
\frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{\Delta t^2} = \frac{c^2}{\Delta x^2} \left( u_{i+1,j} - 2u_{i,j} + u_{i-1,j} \right) \quad (15.142)
$$

Solving for $u_{i,j+1}$ and introducing $\alpha = c^2 \Delta t^2 / \Delta x^2$, we find the simpler form

$$
u_{i,j+1} = \alpha u_{i+1,j} + 2(1 - \alpha)u_{i,j} + \alpha u_{i-1,j} - u_{i,j-1} \quad (15.143)
$$

Evidently, if we know values of $u_{i,j}$ for $j = 0$ and $j = 1$, we can determine values for $j = 2$, then for $j = 3, \ldots$. That is, if we know $u_{i,j}$ along two consecutive horizontal lines in Fig. 15.17, we can step the solution forward in time from line to line by increments of $\Delta t$, going as far as our interest dictates and our patience endures.

As in the partial discretization of the previous paragraph, the boundary conditions help us deal with unknown values that appear when Eq. (15.143) is evaluated at $i = 0$ or $i = N$ by stipulating that

$$u_{0,j} = 0 \quad \text{and} \quad \frac{\partial u}{\partial x}(L, t_j) \approx \frac{u_{N+1,j} - u_{N-1,j}}{2 \Delta x} = 0 \quad \implies \quad u_{N+1,j} = u_{N-1,j} \quad (15.144)
$$

Free of values outside the interval $0 \leq x \leq L$, the equations we use to step the solution to $t_{j+1}$ from knowledge of the solution at previous times then are

$$u_{0,j+1} = 0 \quad (15.145)$$

$$u_{i,j+1} = \alpha u_{i+1,j} + 2(1 - \alpha)u_{i,j} + \alpha u_{i-1,j} - u_{i,j-1}, \quad i = 1, 2, \ldots, N - 1 \quad (15.146)$$

$$u_{N,j+1} = 2\alpha u_{N-1,j} + 2(1 - \alpha)u_{N,j} - u_{N,j-1} \quad (15.147)$$
In their turn, the initial conditions start the process by stipulating that
\[ u_{i,0} = f(x_i) \quad \text{and} \quad u_{i,1} = u_{i,0} + g(x_i) \Delta t \] (15.148)

For consistency with the imposed boundary conditions, we must, of course, require that \( f(0) = 0 \) and \( \frac{dg(x)}{dx}_{x=L} = 0 \). We leave it to Exercise 15.18 to demonstrate that the method described in this paragraph will be unstable unless \( \alpha \leq 1 \).

The discussion at the end of Section 15.2 on discretization and roundoff errors in finite difference methods applies as much in two dimensions as in one. Since solution of the wave equation involves a direct method, however, convergence error as discussed in that earlier section is not here an issue.

We can proceed no further in this example without resorting to a specific coding language, so we postpone further discussion to later sections.

### 15.16.2 The Diffusion Equation

As laid out in Section 15.1.2, the diffusion equation for heat flow in one dimension involves two independent variables, a spatial coordinate \( x \) locating a point in the one-dimensional medium and a time coordinate \( t \). The temperature \( u(x,t) \) at position \( x \) and time \( t \) in the medium satisfies Eq. (15.26),
\[ \frac{\partial u}{\partial t} = \alpha^2 \frac{\partial^2 u}{\partial x^2} \] (15.149)

which is to be solved subject to the initial condition
\[ u(x,0) = f(x) \] (15.150)

that specifies the initial temperature distribution in the medium and to appropriate boundary conditions that specify the way the temperature is controlled at the two ends, say at \( x = 0 \) and \( x = L \). To illustrate both possibilities in a single example, we suppose that, at \( x = 0 \), the temperature is maintained at the value \( T_0 \) and that, at \( x = L \), the medium is insulated so that no heat flow takes place either into or out of the rod at that end, i.e., we suppose that

\[ u(0,t) = T_0 \quad \text{and} \quad \frac{\partial u}{\partial x}(L,t) = 0 \] (15.151)

In one numerical approach to this problem via an FDM, we introduce a set of equally spaced nodes in the spatial dimension but leave the temporal dimension continuous. Suppose we divide the interval \( 0 \leq x \leq L \) into \( N \) elements by \( N+1 \) nodes, each separated from its nearest neighbor(s) by the distance \( \Delta x = L/N \). We denote the coordinates of those nodes by \( x_i = i \Delta x \) with \( i = 0, 1, 2, \ldots, N \). In particular, \( x_0 = 0 \) and \( x_N = L \). Then, we introduce the set of functions \( u_i(t) = u(x_i,t) \), evaluate the PDE of Eq. (15.149) at \( x_i \), and express the spatial derivative at \( x_i \) in terms of finite differences as in Eq. (15.75) to find that
\[ \frac{du_i}{dt} = \frac{\alpha^2}{\Delta x^2} \left( u_{i+1} - 2u_i + u_{i-1} \right) \] (15.152)

The boundary conditions can now be invoked to resolve problems with this equation when \( i = 0 \) and \( i = N \), at which points the equation refers to \( u \) at points outside the domain of the problem, i.e., to \( u_{-1} \) or \( u_{N+1} \), respectively. At \( x = 0 \), the boundary condition implies that \( u_0 = T_0 \), so we can replace Eq. (15.152) at \( i = 0 \) with the equation \( u_0 = T_0 \), though—for the sake of a similar treatment of all nodes—it is more appropriate to regard \( u_0 = T_0 \) as the solution to the initial value problem
\[ \frac{du_0}{dt} = 0 \quad ; \quad u_0(0) = T_0 \] (15.153)
To address Eq. (15.152) at \( i = N \), we look to a central difference approximation of the spatial derivative, finding that
\[
\frac{\partial u}{\partial x}(L, t) \approx \frac{u_{N+1}(t) - u_{N-1}(t)}{2\Delta x} = 0 \quad \implies \quad u_{N+1} = u_{N-1}
\]
(15.154)
Then, we write Eq. (15.152) for \( i = N \), substitute \( u_{N-1} \) for \( u_{N+1} \), and solve the result for \( du_{N}/dt \). In this approach, in which we have discretized only the spatial variable, the problem of solving a PDE in two variables becomes one of solving the coupled set
\[
\begin{align*}
\frac{du_0}{dt} &= 0 \\
\frac{du_i}{dt} &= \frac{\alpha^2}{\Delta x^2} \left( u_{i+1} - 2u_i + u_{i-1} \right), \quad i = 1, 2, 3, \ldots, N - 1 \\
\frac{du_N}{dt} &= 2\frac{\alpha^2}{\Delta x^2} \left( u_{N-1} - u_N \right)
\end{align*}
\]
(15.155 - 15.157)
of ODEs subject to the initial condition
\[
u_i(0) = f(x_i)
\]
(15.158)
where, to reflect properly the conditions imposed on \( u_0 \), we suppose that \( f(x_0) = f(0) = T_0 \) and \( df(x)/dx|_{x=L} = 0 \). By discretizing the spatial variable, we have reduced our problem to a problem of the sort addressed in Chapter 11, and we therefore say no more about it here.

We could go one step further, discretizing also the time variable by introducing a time step \( \Delta t \) and the discrete time instants \( t_j = j \Delta t, \ j = 0, 1, 2, \ldots \). Then, approximating both the spatial and the temporal derivatives by finite differences and introducing the quantities \( u_{ij} = u(x_i, t_j) \), we might evaluate the original PDE of Eq. (15.149) at \( (x, t) = (x_i, t_j) \) and find the fully discretized form
\[
\frac{u_{i,j+1} - u_{i,j}}{\Delta t} = \frac{\alpha^2}{\Delta x^2} \left( u_{i+1,j} - 2u_{i,j} + u_{i-1,j} \right)
\]
(15.159)
Solving for \( u_{i,j+1} \) and introducing \( \gamma = \alpha^2 \Delta t/\Delta x^2 \), we find the simpler form
\[
u_{i,j+1} = \gamma u_{i-1,j} + (1 - 2\gamma)u_{i,j} + \gamma u_{i+1,j}
\]
(15.160)
Evidently, if we know values of \( u_{i,j} \) for \( j = 0 \), we can determine values for \( j = 1 \), then for \( j = 2 \), \ldots. That is, if we know \( u_{i,j} \) along one horizontal line in Fig. 15.18, we can step the solution forward in time from line to line by increments of \( \Delta t \), going as far as our interest dictates and our patience endures.

As in the partial discretization of the previous paragraph, the boundary conditions help us deal with unknown values that appear when Eq. (15.160) is evaluated at \( i = 0 \) or \( i = N \) by stipulating that
\[
u_{0,j} = T_0 \quad \text{and} \quad \frac{\partial u}{\partial x}(L, t_j) \approx \frac{u_{N+1,j} - u_{N-1,j}}{2\Delta x} = 0 \quad \implies \quad u_{N+1,j} = u_{N-1,j}
\]
(15.161)
Free of values outside the interval \( 0 \leq x \leq L \), the equations we use to step the solution to \( t_{j+1} \) from knowledge of the solution at previous times then are
\[
u_{0,j+1} = T_0
\]
(15.162)
\[
u_{i,j+1} = \gamma u_{i-1,j} + (1 - 2\gamma)u_{i,j} + \gamma u_{i+1,j}, \quad i = 1, 2, \ldots, N - 1
\]
(15.163)
\[
u_{N,j+1} = 2\gamma u_{N-1,j} + (1 - 2\gamma)u_{N,j}
\]
(15.164)
In their turn, the initial conditions start the process by stipulating that
\[ u_{i,0} = f(x_i) \quad (15.165) \]
For consistency with the imposed boundary conditions, we must, of course, require that \( f(0) = T_0 \).
We leave it to Exercise 15.19 to demonstrate that the method described in this paragraph will be unstable unless \( \gamma \leq 1/2 \).

The discussion at the end of Section 15.2 on discretization error and roundoff errors in finite difference methods applies as much in two dimensions as in one. Since solution of the diffusion equation involves a direct method, however, convergence error as discussed in that earlier section is not here an issue.

We can proceed no further in this example without resorting to a specific coding language, so we postpone further discussion to later sections.

### 15.16.3 The Laplace Equation

As laid out in Section 15.1.3, the Laplace equation for the steady state temperature distribution in a two-dimensional plate involves two independent spatial variables \( x \) and \( y \) which, together, locate a point in the two-dimensional medium. The temperature \( u(x, y) \) at the point \( (x, y) \) satisfies Eq. (15.27),
\[ \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \quad (15.166) \]
which is to be solved subject to boundary conditions that specify the way the temperature is controlled at the boundaries of the region.\(^{42,43}\)

\(^{42}\)In this case, there are no initial conditions, since we seek only the final steady state temperature, which is determined solely by the (time-independent) boundary conditions. The initial temperature distribution is entirely irrelevant.

\(^{43}\)Note that this approach to the Laplace equation was also discussed in Section 9.3.1, though the example discussed there involved only Dirichlet boundary conditions.
Stipulation of boundary conditions is more complicated for a problem involving two spatial dimensions than for the problems in Sections 15.16.1 and 15.16.2, each of which involved only one spatial dimension. In one spatial dimension, the boundary consists of two points, one at each end of the region. In two spatial dimensions, the boundary is defined by a closed curve—whatever is necessary to bound the region of interest—in the $xy$ plane. Easy application of FDMs is limited to fairly simple geometries in which the boundaries lie along coordinate lines in one or another of the standard coordinate systems. In the present example, we will suppose the region of interest to be a square, each of whose edges has length $L$, and we will suppose that its four corners lie at $(x, y) = (0, 0), (L, 0), (L, L)$, and $(0, L)$ in the $xy$ plane. Then, so as to illustrate both possible types of boundary condition, we will suppose that

$$u(x, 0) = 0 \ ; \ u(x, L) = 100 \ ; \ u(0, y) = 100{y \over L} \ ; \ \partial u / \partial x (L, y) = 0$$

(15.167)

i.e., that the temperature along the line $y = 0$ is maintained at the value 0, that the temperature along the line $y = L$ is maintained at the value 100, that the temperature along the line $x = 0$ rises linearly from 0 at $y = 0$ to 100 at $y = L$, and that the edge along the line $x = L$ is insulated.

The partial discretization of a PDE as illustrated in the previous two sections is applicable only to initial value problems and does not provide a suitable approach to the present boundary value problem. We can approach the present problem successfully only through full discretization of the PDE. Thus, we divide each edge into $N$ segments each of length $\Delta x = L/N$, placing the $(N + 1)^2$ nodes at the points $(x_i, y_j)$, where $x_i = i \Delta x$ ($i = 0, 1, 2, \ldots, N$) and $y_j = j \Delta x$ ($j = 0, 1, 2, \ldots, N$). Then, we introduce the $(N + 1)^2$ values $u_{i,j} = u(x_i, y_j)$, evaluate Eq. (15.166) at the point $(x_i, y_j)$, and express each second partial derivative in terms of finite differences to find that

$$u_{i+1,j} - 2u_{i,j} + u_{i-1,j} \over \Delta x^2 + u_{i,j+1} - 2u_{i,j} + u_{i,j-1} \over \Delta x^2 = 0$$

(15.168)

or, solving for $u_{i,j}$, that

$$u_{i,j} = {1 \over 4} \left( u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} \right)$$

(15.169)

Interestingly, in solutions to Laplace’s equation via an FDM, the value at a particular node is equal to the average of the values at the four nearest neighbors of that node. This geometry is shown in Fig. 15.19.

The boundary conditions can now be invoked to resolve problems with Eq. (15.169) when $i = 0$ or $i = N$ and/or $j = 0$ or $j = N$, at which points the equation refers to one or more nodes lying outside the boundaries of the region in which a solution is sought. We must deal with eight different categories of such nodes:

- nodes on the bottom edge but not at a corner, for which $j = 0$ and $i = 1, 2, \ldots, N - 1$. Along this edge, the boundary condition stipulates that $u_{i,0} = 0$, and we replace Eq. (15.169) with this alternative.
- nodes on the top edge but not at a corner, for which $j = N$ and $i = 1, 2, \ldots, N - 1$. Along this edge, the boundary condition stipulates that $u_{i,0} = 100$, and we replace Eq. (15.169) with this alternative.
- nodes on the left edge but not at a corner, for which $i = 0$ and $j = 1, 2, \ldots, N - 1$. Along this edge, the boundary condition stipulates that $u(0, y) = 100y/L$, or $u_{0,j} = 100y_j/L$, and we replace Eq. (15.169) with this alternative.
- nodes on the right edge but not at a corner, for which $i = N$ and $j = 1, 2, \ldots, N - 1$. This edge is to be insulated, i.e., $\partial u / \partial x = 0$ on that edge. We use a central difference approximation to the derivative to find that

$$\left. \partial u / \partial x (x, y) \right|_{x = L} = 0 \implies {u_{N+1,j} - u_{N-1,j} \over 2 \Delta x} = 0 \implies u_{N+1,j} = u_{N-1,j}$$

(15.170)
Figure 15.19: Geometry for full discretization of the Laplace equation. Equation 15.169 involves the points marked with solid figures and has been solved to express the solution at the (one) point marked with a solid square in terms of the solutions at the (four) points marked with solid circles.

For these nodes, Eq. (15.169) becomes

\[
\begin{align*}
    u_{N,j} &= \frac{1}{4} \left( u_{N+1,j} + u_{N-1,j} + u_{N,j+1} + u_{N,j-1} \right) \\
    &= \frac{1}{4} \left( 2u_{N-1,j} + u_{N,j+1} + u_{N,j-1} \right) 
\end{align*}
\]

and, as long as \( j \) stays in the specified range, this equation no longer involves points outside the region defined by the boundaries.

- the node at the lower left corner for which \( i = 0 \) and \( j = 0 \). This point lies on two of the bounding edges. Fortunately, in the present case, the boundary conditions on those two edges are consistent, and we simply replace Eq. (15.169) with \( u_{0,0} = 0 \). In fact, however, this point will never enter into any equation we will need to consider, so the value we assign at this one point is of no consequence. Indeed, in some problems, there may be a discontinuity in the temperature at an isolated point such as this one. Conveniently, the method we have adopted is not upset by such a discontinuity.

- the node at the upper left corner, for which \( i = 0 \) and \( j = N \). This point is similar to the point \((i,j) = (0,0)\), even to the consistency of the values on the two edges to which it belongs. We set \( u_{0,N} = 100 \).

- the node at the lower right corner, for which \( i = N \) and \( j = 0 \). Here, the boundary condition on the lower edge dictates that we should set \( u_{N,0} = 0 \). Applied to this node, however, Eq. (15.171) with \( j = 0 \) suggests that we should require that

\[
    u_{N,0} = \frac{1}{4} \left( 2u_{N-1,0} + u_{N,1} + u_{N,-1} \right) 
\]

Consistency with the presumed value \( u_{N,0} = 0 \) and the known value \( u_{N-1,0} = 0 \) then implies that we should expect that

\[
    u_{N,-1} = -u_{N,1} 
\]

a result that we might also infer from symmetries if we saw the problem of interest as the upper half of a larger problem obtained by reflecting the problem we are addressing into the
region \(-L < y < 0\), making the temperature on the bottom edge of that larger region \(-100\). We conclude that taking \(u_{N,0} = 0\) is appropriate and justified.

- the node at the upper right corner, for which \(i = N\) and \(j = N\). This point is similar to the point \((i, j) = (N, 0)\) and, without further argument, we accept the replacement \(u_{N,N} = 100\).

In short, the equations we seek to solve, now involving no quantities at points outside the boundaries of the problem, are

\[
\begin{align*}
    u_{i,j} &= 1/4 (u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1}) ; \quad 1 \leq i, j \leq N - 1 \\
    u_{i,0} &= 0 ; \quad 1 \leq i \leq N - 1 \\
    u_{i,N} &= 100 ; \quad 1 \leq i \leq N - 1 \\
    u_{0,j} &= 100 \frac{y_j}{L} ; \quad 1 \leq j \leq N - 1 \\
    u_{N,j} &= 1/4 (2u_{N-1,j} + u_{N,j+1} + u_{N,j-1}) ; \quad 1 \leq j \leq N - 1 \\
    u_{0,0} &= 0 \\
    u_{0,N} &= 100 \\
    u_{N,0} &= 0 \\
    u_{N,N} &= 100
\end{align*}
\]

We can proceed no further in this example without resorting to a specific coding language, so we postpone further discussion to later sections. We do note here, however, that iterative methods will almost always be used to solve this set of equations. Consequently (and in contrast to the situation with the wave and diffusion equations), we will have to pay attention not only to discretization and roundoff errors but also to convergence error.

### 15.22 Using C to Solve 2D PDEs via an FDM

To complete the solution of the problems laid out in Section 15.16, we must choose a specific language for the development of explicit coding to implement the algorithms described. In this section, we illustrate that process in C.

#### 15.22.1 The Wave Equation

##### 15.22.1.1 The C Coding

The final step in addressing the example laid out in Section 15.16.1 is to set up and solve Eqs. (15.145)–(15.147). We begin by including necessary system files, declaring necessary variables, requesting input of all necessary parameters, and calculating \(dx\) and \(alpha\) with the statements

```c
#include <stdio.h>        /* For I/O */
#include <math.h>         /* For math functions */
float dt, c, l, dx, alpha;    /* For parameters */
int n, nt;                /* For parameters */
printf( "\nNumber of segments: "); scanf ( "%d", &n );
printf( "Time step: "); scanf ( "%f", &dt );
```

printf( "Number of time steps: "); scanf ( "%d", &nt );
printf( "Speed of propagation: "); scanf ( "%f", &c );
printf( "Length of string: "); scanf ( "%f", &l );
dx = l/n;
alpha = pow(c,2)*pow(dt,2)/pow(dx,2);

Prudence dictates the wisdom of displaying the parameter alpha and trapping unacceptable values
with the statements

#include <stdlib.h> /* To define exit */
printf( " alpha = %8.3f\n", alpha );
if alpha > 1.0
{ printf("\n "alpha > 1; execution halted\n")
  exit(0);
}

to halt execution if the parameter $\alpha$ has a value that guarantees an unstable—and hence inaccurate—solution.

We next recognize that we will need one one-dimensional array, call it $x$, for the values of the
independent variable at which a solution is generated and three one-dimensional arrays, call them $u_1$, $u_2$, and $u_3$, for the past solution, the current solution, and the next solution, respectively. To
provide the flexibility to specify the refinement of elements without having to rewrite the program,
we elect to dimension several needed arrays at execution time, To allocate memory at execution
time, we must invoke the C function malloc which (along with exit) is defined in the include file stdlib.h. Then, once we know the required sizes of the arrays, we allocate the necessary space explicitly.\footnote{Technically, once we no longer need the dynamically allocated variable, we should use the function free() to free up the allocated space for other uses. Since we here are not taxing memory and since the space will be freed as soon as the program returns control to the operating system, we need not free the space explicitly in the program.}
The statements

float *x, *u1, *u2, *u3; /* Define pointers to solutions */
x = (float*)malloc((n+1)*sizeof(float));
u1 = (float*)malloc((n+1)*sizeof(float));
u2 = (float*)malloc((n+1)*sizeof(float));
u3 = (float*)malloc((n+1)*sizeof(float));

int i; /* For loop counter */
for(i=0; i<=n; i++)
{ u1[i]=0.0; u2[i]=0.0; u3[i]=0.0;
  x[i] = dx*i;

create these variables, declare an index for loops, initialize $u_1$, $u_2$, and $u_3$, and establish values for
the $x$ coordinates at which solutions will be generated.

Next, before coding the loop that will generate the solution, step by step, we must set the values
in $u_1$ to reflect the initial displacement of the string, set the values in $u_2$ to reflect the impact of the
initial velocity on the motion during the first time step, and display $u_1$ and $u_2$. To be specific, let
us suppose that, initially, the string is displaced in the shape of a single hump of a sine wave, but
only in the middle quarter of its length, and is undisplaced over the first three-eighths and the last
three eighths of its length. Thus, we suppose that

\[
 f(x) = \begin{cases} 
 0 & 0 \leq x \leq \frac{3}{8} L \\ 
 1 + \cos \left( \frac{8\pi}{L} \left( x - \frac{L}{2} \right) \right) & \frac{3}{8} L \leq x \leq \frac{5}{8} L \\ 
 0 & \frac{5}{8} L \leq x \leq L 
\end{cases} 
\quad (15.175)
\]

which results in a smooth transition from zero displacement outside the center one quarter of the string and the sinusoidal displacement in that interval. Further, we suppose that the string is released from rest so that

\[
 g(x) = 0 
\quad (15.176)
\]
or, equivalently—see Eq. (15.148)—\(u_{i,0} = u_{i,1}\) or \(u_2 = u_1\). The coding that will impose these initial conditions, display the initial solution, and then display the solution after the first time step then is\(^{45}\)

```c
#define pi 3.1415926535
float t, b;
t = 0;
b = 8.0*pi/l;
for(i = 3*n/8 + 1; i<=5*n/8)
    u1[i] = 1.0 + cos(b*(x[i]-l.0/2.0))
printf( "%7.3f\n", t );
for( i=0; i<=n; i++)
    printf("%7.3f", u1[i]);
printf( "\n " )
for(i=0; i<=n; i++)
    u2[i]=u1[i];
t = t + dt;
printf( "%7.3f\n", t );
for( i=0; i<=n; i++)
    printf("%7.3f", u2[i]);
printf( "\n " )
```

We add a definition of the parameter \(\pi\), and initialization and incrementation of a time variable, and a display of that time variable to our coding as well.

Now, we are ready to code the algorithm that uses Eqs. (15.145)–(15.147) to advance the solution, step by step. Appropriate coding, or at least a first pass at such coding, might be\(^{46}\)

```c
int j;
for(j=2; j<nt; j++)
{ u3[0]=0.0;
  for(i=1; i<n; i++)
    \{ u3[i] = alpha*u2[i+1] + 2.0*(1.0-alpha)*u2[i] + alpha*u2[i-1] - u1[i];
    u3[n] = 2.0*alpha*u2[n-1]+2.0*(1.0-alpha)*u2[n] - u1[n];
    t = dt*j;
    printf( "%7.3f\n", t );
\}
```

\(^{45}\)The calculation of the range of \(i\) to be used is complicated by one issues. Because of integer arithmetic, the quantities \(3*n/8\) and \(5*n/8\) will be truncated, but we really want the lower limit to be raised rather than truncated. The upper limit can be truncated. Thus, we add 1 to the lower limit.

\(^{46}\)To avoid cumulative roundoff in the calculation of times, we elect to abandon repeated incrementation in favor of direct calculation of each time from the loop index.
Here, each pass through the outermost loop advances the solution by one time step. Within that loop, we (1) construct the solution at the next time instant by exploiting Eqs. (15.145)–(15.147), (2) increment the time variable and display both the time and the solution, and (3) shift the values to prepare for the next pass through the loop. A more fully commented command file containing these statements—and a few others to be discussed in a moment—and collecting all variable declarations at the beginning of the main program is named fdmwave1d.c, is listed in Appendix 15.D.1, and can be copied from the directory $HEAD/cc.

Two problems emerge when we run the coding developed to this point with a trial set of values, say \( n = 100, \) \( dt = 0.1, \) \( nt = 20, \) \( c = 0.5, \) and \( l = 10 \) (for which \( \alpha = 0.25 \)). First, there is so much output on the screen that interpretation is difficult. Second, even with 20 time steps, the solution is not advanced very far.

We ignore the first of these problems because we will ultimately recast the program to write the solution into a file and actually examine that solution graphically by importing that file into another program. From that perspective, the confusion engendered by the solution on the screen in the current version of the program is not an issue.

We can, of course, fix the second problem simply by requesting a larger number of time steps or enlarging the time step (or both—though we must be careful not to increase \( \alpha \) beyond 1.0). As the number of time steps requested increases—and, perhaps, the time interval between steps decreases, however, the program as we now have it, which displays the solution at every step of the way, will produce increasing volumes of output. With the more accurate solutions (numerous segments, short time interval between generated solutions, many time steps to be computed), we would be wise to introduce a mechanism for suppressing the display of many of the generated solutions. To do so, we add at the beginning the statement

```c
int nf;
printf( "Plot frequency: "); scanf ( "%d", &nf );
```

to request the number of solutions whose display should be suppressed after a particular solution has been displayed and, in addition, we modify the critical `printf` statements to make the display conditional upon \( j \)—the loop index—being a multiple of \( nf \). In effect, we simply replace the `printf` statement everywhere it appears with the conditional statement

```c
if (nf*(j/nf) == j)
{   printf( "\%7.3f\n", t );
    for( i=0; i<=n; i++)
        printf("\%7.3f", u1[i]);
    printf( "\n" );
}
```

Finally, we wish to modify the program so that the output is written not to the screen but to a file, thereby facilitating its input into a graphical visualization program for careful examination. Essentially, we need to open the file on an appropriate channel before writing anything into it, change all `printf` statements to direct their output to that channel, and close the file after the last item is

\[47\] Since both \( nf \) and \( j \) are integers, the division \( (j/nf) \) will be an integer division. Thus, \( nf*(j/nf) \) will be equal to \( j \) only when \( j \) is a multiple of \( nf \).

\[48\] Of course, \( u1 \) has to be replaced by \( u2 \) or \( u3 \) in some of the statements.
written to it. In addition, we elect to write \( n, c, l \), and \( \alpha \) in a single line at the beginning of the file so that that information is also available to the graphical visualization program. Thus, we add the statements

```c
FILE *fptr; /* For file pointer */
fptr = fopen( "fdmwave1d_c.dat", "w" );
fprintf( fptr, "%d %f %f %f \n", n, c, l, alpha );
```

change all `printf` statements to direct their output to the file with the command `fprintf`, and add the statement

```c
fclose( fptr );
```

after the last `fprintf` statement. The complete program is listed in Appendix 15.D.1.

### 15.22.1.2 Running the C Program

We would compile and link this program with the simple statement

```c
cc -o fdmwave1d.xc fdmwave1d.c
```

to the operating system.\(^{49}\)

While it will certainly take a bit of trial and error, including some bouncing back and forth between the program generating the solution and the visualization program used to examine the solution, we might ultimately run the program with the controlling values

```
./fdmwave1d.xc
Number of segments: 100
Time step: 0.1
Number of time steps: 800
Speed of propagation: 0.5
Length of string: 10.0
Plot frequency: 5
\alpha = 0.2500000
```

generating the file `fdmwave1d_c.dat`.

### 15.22.1.3 Displaying the Solution Graphically with IDL

To display this solution in IDL, we would first have to open the file and import not only the first line of data \( n, c, l \), and \( \alpha \) but also the first time and the solution at that first time. The statements

```idl
IDL> openr, 1, 'fdmwave1d_c.dat'
IDL> readf, 1, n, c, l, alpha & n = fix(n)
IDL> u = fltarr(n+1)
IDL> readf, 1, t, u
```

will accomplish that task. Then, using the information supplied in the first line, we use the statement

\(^{49}\)Here and in what follows in this section, we illustrate the statements to be used in a UNIX operating system. Your *Local Guide* will explain the statements needed in your operating system.
Figure 15.20: Representative frames in motion of a string. This graph was produced with IDL from a data file generated by a C program.

IDL> x = l*findgen(n+1)/float(n)

to calculate the values of the independent variable at which nodes were placed along the string, and we use the statement

IDL> !y.range=[-2.0,2.0]

to establish a fixed vertical scaling for all graphs. Then, a quick graph of the initial shape can be generated with the statement

IDL> plot, x, u, thick=3.0, title='t = ' + string(t, FORMAT='(F10.3)' )

The resulting graph is shown in the upper left frame of Fig. 15.20.

We could create an animated picture of the string’s behavior in time by continuing to read from the file and display the output with the statements\(^{50}\)

IDL> while not eof(1) do begin $
IDL> \text{readf, 1, t, u &}$
IDL> \text{plot, x, u, thick=3.0, title='t = ' + string(t, FORMAT='(F10.3)' ) &}$

\(^{50}\)Note that only one \texttt{end} statement (\texttt{endwhile}) is necessary in this context.
15.22. USING C TO SOLVE 2D PDES VIA AN FDM

IDL>  wait, 0.1 & $
IDL>  endwhile
IDL>  close, 1

This output also agrees with that displayed in Fig. 15.20.

15.22.1.4 Displaying the Solution Graphically with MATLAB

To display this solution in MATLAB, we would first have to (1) open the file, (2) import the first line of data (N, C, AL, and ALPHA), and (3) preserve for later use the current position of the file pointer (which is the beginning of the “real” data in the file). The statements

```matlab
id = fopen( 'fdmwave1d_c.dat', 'r' );
n = fscanf( id, '%d', 1 );
c = fscanf( id, '%f', 1 );
l = fscanf( id, '%f', 1 );
alpha = fscanf( id, '%f', 1 );
position = ftell( id );
```

will accomplish those tasks. Then, using the information supplied in the first line, we use the statement

```matlab
x = [ 0.0 : l/n : l ];
```

to calculate the values of the independent variable at which nodes were placed along the string, and we use the statements

```matlab
t = fscanf( id, '%f', 1 ); % Prime t before entering loop
while ( ~feof( id ) )
    u = transpose( fscanf( id, '%f', n+1 ) );
    plot( x, u, 'Color', 'black', 'LineWidth', 4 );
    set( gca, 'YLim', [-2.0,2.0] );
    title( ['t = ' num2str( t, '%10.4f' )], 'FontSize', 20 );
    pause( 0.1 );
    t = fscanf( id, '%f', 1 ); % Read blank line at end, triggering EOF
end
```

This output also agrees with that displayed in Fig. 15.21.
Figure 15.21: Representative frames in motion of a string. This graph was produced with MATLAB from a data file generated by a C program.

15.22.1.5 Displaying the Solution Graphically with OCTAVE

To display this solution in OCTAVE, we would first have to (1) open the file, (2) import the first line of data (N, C, AL, and ALPHA), and (3) preserve for later use the current position of the file pointer (which is the beginning of the “real” data in the file). The statements

```octave
id = fopen('fdmwave1d_c.dat', 'r');
n = fscanf(id, '%d', 1);
c = fscanf(id, '%f', 1);
l = fscanf(id, '%f', 1);
alpha = fscanf(id, '%f', 1);
position = ftell(id);
```

will accomplish those tasks. Then, using the information supplied in the first line, we use the statement

```octave
x = [ 0.0 : l/n : l ];
```

to calculate the values of the independent variable at which nodes were placed along the string, and we use the statements
Figure 15.22: Representative frames in motion of a string. This graph was produced with OCTAVE from a data file generated by a C program.

```c
// Example C code to read data and plot

double t = fscanf( id, '%f', 1 );
double u = transpose( fscanf( id, '%f', n+1 ) );
plot( x, u, 'Color', 'black', 'LineWidth', 4 );
set( gca, 'YLim', [-2.0,2.0] )
title( ['t = ', num2str( t, '%10.4f' )], 'FontSize', 20 )
```

to read the first time and the solution at that first time from the file and generate a quick graph of the initial shape of the string. This picture is in complete agreement with the upper left panel in Fig. 15.22.

We could create an animated picture of the string’s behavior in time by continuing to read from the file, displaying each solution in turn. We restore the file pointer to the start of the “real” data with the statement

```c
status = fseek( id, position, 'bof' );
```

Then, we can code a loop to generate the desired animation with the statements

```c
// Example C code for animation

t = fscanf( id, '%f', 1 );  // Prime t before entering loop
while ( !feof( id ) )
    u = transpose( fscanf( id, '%f', n+1 ) );
    plot( x, u, 'Color', 'black', 'LineWidth', 4 );
```
set( gca, 'YLim', [-2.0,2.0] );
title( ['t =', num2str( t, '%10.4f' )], 'FontSize', 20 );
pause( 0.1 );
t = fscanf( id, '%f', 1 ); # Read blank line at end, triggering EOF
end
status = fclose( id );

This output also agrees with that displayed in Fig. 15.22.

15.22.1.6 Displaying the Solution Graphically with PYTHON

To display this solution in PYTHON, we would first have to open the file and import not only
the first line of data (n, c, l, and alpha) but also the first time and the solution at that first time. The
statements

```python
f = open( 'fdmwave1d_c.dat', 'r' )
tmp = [float(x) for x in f.readline().split()]
n = int(tmp[0]); c = tmp[1]; l = tmp[2]; alpha = tmp[3]
```

will accomplish that task. Then, using the information supplied in the first line, we use the state-
ments

```python
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace( 0.0,1.0,n+1)
```
to import needed modules and calculate the values of the independent variable at which nodes were
placed along the string, and we use the statements

```python
t = float( f.readline() )
u = [float(tmp) for tmp in f.readline().split()]
plt.plot( x,u, color='black', linewidth=3 )
plt.title( 't = '+str(t) )
plt.ylim( [-2.0,2.0] )
plt.show()
```
to read the first time and the solution at that first time from the file and then generate a quick
graph of the initial shape of the string. This picture is in complete agreement with the upper left
panel in Fig. 15.23.

We could create an animated picture of the string's behavior in time by continuing to read from
the file, displaying each solution in turn. We restore the file pointer to the beginning of the file,
reopen the file, and reread the initial line in the file, and recreate the coordinates along the x axis
with the statements

```python
f.close()
f = open( 'fdmwave1d_c.dat', 'r' )
tmp = [float(x) for x in f.readline().split()]
n = int(tmp[0]); c = tmp[1]; l = tmp[2]; alpha = tmp[3]
x = np.linspace( 0.0,1.0,n+1)
```

Then, we can code a loop to generate the desired animation with the statements
Figure 15.23: Representative frames in motion of a string. This graph was produced with PYTHON from a data file generated by a C program.

```python
tp = f.readline()  # Read next line as string
while ( tp != "" ):  # Loop while not at end of file
    t = float(tp)  # Convert tp to numerical value
    # Read next solution
    u = [float(tmp) for tmp in f.readline().split()]  # Read next solution
    plt.clf()  # Clear figure window
    plt.plot( x,u, color='black', linewidth=3 )  # Plot next solution
    plt.title( 't = ' + str(t) )  # Add title
    plt.ylim( [-2.0,2.0] )  # Standardize y axis range
    plt.pause( 0.1 )  # Display graph
    tp = f.readline()  # Read next line as string
f.close()
plt.show()
```

When run interactively, all of the usual reports of progress in the plotting will be displayed on the screen. If the program is stored in a file and run as a command file, all of that dialog will be suppressed. The resulting output in either case agrees with that displayed in Fig. 15.23.
15.22.2 The Diffusion Equation

15.22.2.1 The C Coding

The final step in addressing the example laid out in Section 15.16.2 is to set up and solve Eqs. (15.162)–(15.164). We begin by including necessary system files, declaring necessary variables, requesting input of all necessary parameters, and calculating $dx$ and $ggamma$ with the statements

```c
#include <stdio.h>  /* For I/O */
#include <math.h>   /* For math functions */

float dt, alpha, l, dx, ggamma; /* For parameters */
int n, nt; /* For parameters */

printf( "Number of segments: "); scanf ( "%d", &n );
printf( "Time step: "); scanf ( "%f", &dt );
printf( "Number of time steps: "); scanf ( "%d", &nt );
printf( "Value of alpha: "); scanf ( "%f", &alpha );
dx = l.0/n;
ngamma = pow(alpha,2)*dt/pow(dx,2);
```

Prudence dictates the wisdom of displaying the parameter $\alpha$ and trapping unacceptable values with the statements

```c
#include <stdlib.h>  /* To define exit */

#include <stdio.h>  /* To define exit */
float *x, *u1, *u2; /* Pointers to solutions */

x = (float*)malloc((n+1)*sizeof(float));
u1 = (float*)malloc((n+1)*sizeof(float));
u2 = (float*)malloc((n+1)*sizeof(float));
```

The variable $\gamma$ is reserved in C, so we use the variable $ggamma$ for $\gamma$.

Technically, once we no longer need the dynamically allocated variable, we should use the function `free()` to free up the allocated space for other uses. Since we here are not taxing memory and since the space will be freed as soon as the program returns control to the operating system, we need not free the space explicitly in the program.
int i; /* For loop counter */
for(i=0; i<=n; i++)
{ u1[i]=0.0; u2[i]=0.0;
x[i] = dx*i;
}

create these variables, declare an index for loops, initialize \( u_1 \) and \( u_2 \) and establish values for the \( x \) coordinates at which solutions will be generated.

Next, before coding the loop that will generate the solution, step by step, we must set the values in \( u_1 \) to reflect the initial temperature distribution in the rod and display \( u_1 \). To be specific, let us suppose that, initially, the temperature exhibits a single hump of a sine wave, but only in the middle quarter of its length, and the temperature remains zero over the first three-eighths and the last three eighths of its length. Thus, we suppose that

\[
f(x) = \begin{cases} 
0 & 0 \leq x \leq \frac{3}{8}L \\
1 + \cos \frac{8\pi}{L} \left( x - \frac{L}{2} \right) & \frac{3}{8}L \leq x \leq \frac{5}{8}L \\
0 & \frac{5}{8}L \leq x \leq L 
\end{cases}
\] (15.177)

which results in a smooth transition from zero temperature outside the center one quarter of the rod and the sinusoidal variation in the temperature in that interval. The coding that will impose these initial conditions and display the initial temperature distribution is\textsuperscript{53}

```c
#define pi 3.1415926535
float t, b;
t = 0;
b = 8.0*pi/l;
for(i = 3*n/8 + 1; i<=5*n/8)
    u1[i] = 1.0 + cos(b*(x[i]-l/2.0));
printf( "%7.3f\n", t );
for( i=0; i<=n; i++)
    printf("%7.3f", u1[i]);
printf("\n");
```

We add a definition of the parameter \( \pi \), initialization of a time variable, and a display of that time variable to our coding as well.

Now, taking the temperature at \( x = 0 \) to be zero \( (T_0 = 0) \), we are ready to code the algorithm that uses Eqs. (15.162)–(15.164) to advance the solution, step by step. Appropriate coding, or at least a first pass at such coding, might be

```c
int j;
for(j=1; j<=nt; j++)
{ u2[0]=0.0;
  for(i=1; i<n; i++)
      u2[i] = ggamma*u1[i-1] + (1.0-2.0*ggamma*u1[i] + ggamma*u1[i+1];
  u2[n] = 2*ggamma*u1[n-1] + (1.0-2.0*ggamma)*u1[n];
  printf( "%7.3f\n", dt*j);
}
```

\textsuperscript{53}The calculation of the range of \( i \) to be used is complicated by one issue. Because of integer arithmetic, the quantities \( 3n/8 \) and \( 5n/8 \) will be truncated, but we really want the lower limit to be raised rather than truncated. The upper limit can be truncated. Thus, we add 1 to the lower limit.
Here, each pass through the outermost loop advances the solution by one time step. Within that loop, we (1) construct the solution at the next time instant by exploiting Eqs. (15.162)–(15.164), (2) recalculate the time variable and display both the time and the solution, and (3) shift the values to prepare for the next pass through the loop. A more fully commented command file containing these statements—and a few others to be discussed in a moment—is named fdmwave1d.c, is listed in Appendix 15.D.2, and can be copied from the directory $HEAD/cc.

Two problems emerge when we run the coding developed to this point with a trial set of values, say $n = 100, \, dt=0.1, \, nt=25, \alpha = 0.1$, and $l=10$ (for which $ggamma = 0.1$). First, there is so much output on the screen that interpretation is difficult. Second, even with 25 time steps, the solution is not advanced very far.

We ignore the first of these problems because we will ultimately recast the program to write the solution into a file and actually examine that solution graphically by importing that file into another program. From that perspective, the confusion engendered by the solution on the screen in the current version of the program is not an issue.

We can, of course, fix the second problem simply by requesting a larger number of time steps or enlarging the time step (or both—though we must be careful not to increase $ggamma$ beyond 0.5). As the number of time steps requested increases—and, perhaps, the time interval between steps decreases, however, the program as we now have it, which displays the solution at every step of the way, will produce increasing volumes of output. With the more accurate solutions (numerous segments, short time interval between generated solutions, many time steps to be computed), we would be wise to introduce a mechanism for suppressing the display of many of the generated solutions. To do so, we add at the beginning the statement

```c
int nf;
printf( "Plot frequency: "); scanf ( "%d", &nf );
```

to request the number of solutions whose display should be suppressed after a particular solution has been displayed and, in addition, we modify the critical `printf` statements in the final loops to make the display conditional upon $j$—the loop index—being a multiple of $nf$. In effect, we simply replace the most of the `printf` statements with the conditional statement:\footnote{Since both $nf$ and $j$ are integers, the division $(j/nf)$ will be an integer division. Thus, $nf*(j/nf)$ will be equal to $j$ only when $j$ is a multiple of $nf$.}

```c
if (nf*(j/nf) == j)
{ printf( "%7.3f\n", j*dt );
  for( i=0; i<n; i++)
    printf("%8.4f", u2[i]);
  printf( "\n" );
}
```

(See the full listing in Appendix 15.D.2 for the details.)

Finally, we wish to modify the program so that the output is written not to the screen but to a file, thereby facilitating its input into a graphical visualization program for careful examination. Essentially, we need to open the file with a suitable ID before writing anything into it, change all `printf` statements to direct their output to that file, and close the file after the last item is written.
to it. In addition, we elect to write \( n \), \( \alpha \), 1, and \( \gamma \) in a single line at the beginning of the file so that that information is also available to the graphical visualization program. Thus, we add the statements

```c
FILE *fptr; /* For file pointer */
fptr = fopen("fdmdiffus1d_c.dat", "w");
fprintf(fptr, "%d %f %f %f \n", n, alpha, l, gamma);
```

change all `printf` statements to direct their output to the file with the command `fprintf` and add the statement

```c
fclose(fptr);
```

after the last `fprintf` statement. The complete program is listed in Appendix 15.D.2.

### 15.22.2.2 Running the C Program

We would compile and link this program with the simple statement

```
f77 -o fdmdiffus1d.xc fdmdiffus1d.c
```

to the operating system.\(^5\)

While it will certainly take a bit of trial and error, including some bouncing back and forth between the program generating the solution and the visualization program used to examine the solution, we might ultimately run the program with the controlling values

```
./fdmdiffus1d.xc
Number of segments: 100
Time step: 0.4
Number of time steps: 4400
Value of alpha: 0.1
Length of string: 10
Plot frequency: 50
\gamma = 0.400
```

to generate the file `fdmdiffus1d_c.dat` containing data on the evolution of the initial temperature distribution adequately into the future.

### 15.22.2.3 Displaying the Solution Graphically with IDL

To display this solution in IDL, we would first have to open the file and import not only the first line of data (\( n \), \( \alpha \), 1, and \( \gamma \)) but also the first time and the solution at that first time. The statements

```
IDL> openr, 1, 'fdmdiffus1d_c.dat'
IDL> readf, 1, n, alpha, l, gamma & n = fix(n)
IDL> u = fltarr(n+1)
IDL> readf, 1, t, u
```

\(^5\)Here and in what follows in this section, we illustrate the statements to be used in a UNIX operating system. Your Local Guide will explain the statements needed in your operating system.
Figure 15.24: Temperatures in a rod at various times as it approaches equilibrium. Note that the vertical scale after the graph for \( t = 0 \) s has been expanded the better to show the gradual evolution of the temperature distribution. This graph was produced with IDL from a data file generated by a C program.

IDL> x = l*findgen(n+1)/float(n)

will accomplish that task. Then, using the information supplied in the first line, we use the statement

IDL> !y.range=[0.0,2.0]

Note that only one end statement (endwhile) is necessary in this context.
15.22. USING C TO SOLVE 2D PDES VIA AN FDM

IDL> while not eof(1) do begin $
IDL> \text{readf, 1, t, u} & \nonumber$
IDL> \text{plot, x, u, thick=3.0, title='t = ' + string(t, FORMAT='(F10.3)') &} \nonumber$
IDL> \text{wait, 0.1} & \nonumber$
IDL> \text{endwhile}$
IDL> close, 1

This output also agrees with that displayed in Fig. 15.24.

15.22.2.4 Displaying the Solution Graphically with MATLAB

To display this solution in MATLAB, we would first have to open the file, import the first line of
data (n, alpha, l, and gamma), and preserve for later use the current location of the file pointer
(which is at the beginning of the “real” data in the file). The statements

$id = \text{fopen( 'fdmdiffus1d_c.dat', 'r' );}$
$n = \text{fscanf( id, '%d', 1 );}$
$\alpha = \text{fscanf( id, '%f', 1 );}$
$l = \text{fscanf( id, '%f', 1 );}$
$\gamma = \text{fscanf( id, '%f', 1 );}$
$\text{position = ftell( id );}$

will accomplish those tasks. Then, using the information supplied in the first line, we use the
statement

$x = [ 0.0 : l/n : l ];$

to calculate the values of the independent variable at which nodes were placed along the string, and
we use the statements

$t = \text{fscanf( id, '%f', 1 );}$
$u = \text{transpose( fscanf( id, '%f', n+1 );}$
$\text{plot( gca, 'YLim', [0.0,2.0] ) }$
$\text{title( ['t = ' num2str( t, '%10.4f' )], 'FontSize', 20 )}$

will read the first time and the solution at that first time from the file and generate a quick graph of
the initial shape of the string. The resulting graph is shown in the upper left frame of Fig. 15.25.

We could create an animated picture of the evolution of the temperature in this rod in time by
continuing to read from the file, displaying each solution in turn. We restore the file pointer to the
start of the “real” data with the statement

\text{status = fseek( id, position, 'bof');}

Then, we can code a loop to generate the desired animation with the statements

$t = \text{fscanf( id, '%f', 1 );}$  % Prime t before entering loop
while ( `feof( id )` )
   $u = \text{transpose( fscanf( id, '%f', n+1 );}$
   $\text{plot( x, u, 'Color', 'black', 'LineWidth', 4 );}$
   
$^{57}$For compatibility with MATLAB, we here use $\gamma$, since $\gamma$ is reserved in MATLAB.
Figure 15.25: Temperatures in a rod at various times as it approaches equilibrium. Note that the vertical scale after the graph for $t = 0$ s has been expanded the better to show the gradual evolution of the temperature distribution. This graph was produced with MATLAB from a data file generated by a C program.

```
set( gca, 'YLim', [0.0,2.0] );
title( ['t =', num2str( t, '%10.4f' )], 'FontSize', 20 );
pause( 0.1 );
t = fscanf( id, '%f', 1 ); % Read blank line at end, triggering EOF
end
status = fclose( id );
```

This output also agrees with that displayed in Fig. 15.25.

### 15.22.2.5 Displaying the Solution Graphically with OCTAVE

To display this solution in OCTAVE, we would first have to open the file, import the first line of data ($n$, $\alpha$, $l$, and $\gamma$), and preserve for later use the current location of the file pointer (which is at the beginning of the “real” data in the file). The statements\(^{58}\)

```
id = fopen( 'fdmdiffusid_c.dat', 'r' );
n = fscanf( id, '%d', 1 );
alpha = fscanf( id, '%f', 1 );
l = fscanf( id, '%f', 1 );
```

\(^{58}\)For compatibility with MATLAB, we here use $g\gamma$, since $\gamma$ is reserved in MATLAB.
Figure 15.26: Temperatures in a rod at various times as it approaches equilibrium. Note that the vertical scale after the graph for $t = 0.0$ s has been expanded the better to show the gradual evolution of the temperature distribution. This graph was produced with OCTAVE from a data file generated by a C program.

```
gamma = fscanf( id, '%f', 1 );
position = ftell( id );
```

will accomplish those tasks. Then, using the information supplied in the first line, we use the statement

```
x = [ 0.0 : l/n : l ];
```

to calculate the values of the independent variable at which nodes were placed along the string, and we use the statements

```
t = fscanf( id, '%f', 1 );
u = transpose( fscanf( id, '%f', n+1 ) );
plot( x, u, 'Color', 'black', 'LineWidth', 4 )
set( gca, 'YLim', [0.0,2.0] )
title( ['t = ' num2str( t, '%10.4f' )], 'FontSize', 20 )
```

to read the first time and the solution at that first time from the file and generate a quick graph of the initial shape of the string. The resulting graph is shown in the upper left frame of Fig. 15.26.

We could create an animated picture of the evolution of the temperature in this rod in time by continuing to read from the file, displaying each solution in turn. We restore the file pointer to the start of the “real” data with the statement
status = fseek( id, position, 'bof' );

Then, we can code a loop to generate the desired animation with the statements

\[
t = fscanf( id, '%f', 1 ); \quad \# \text{Prime } t \text{ before entering loop}
\]

\[
\text{while ( !feof( id ) )}
\]

\[
\begin{align*}
\quad & u = \text{transpose( fscanf( id, '%f', n+1 ) );} \\
\quad & \text{plot( x, u, 'Color', 'black', 'LineWidth', 4 );} \\
\quad & \text{set( gca, 'YLim', [0.0,2.0] );} \\
\quad & \text{title( ['t =', num2str( t, '%10.4f' )], 'FontSize', 20 );} \\
\quad & \text{pause( 0.1 );} \\
\quad & t = fscanf( id, '%f', 1 ); \quad \# \text{Read blank line at end, triggering EOF}
\end{align*}
\]

\[
\text{end}
\]

\[
\text{status = fclose( id );}
\]

This output also agrees with that displayed in Fig. 15.26.

### 15.22.2.6 Displaying the Solution Graphically with PYTHON

To display this solution in PYTHON, we would first have to open the file and import the first line of data containing \( (n, \alpha, \lambda, \text{ and } \gamma) \). The statements

\[
\begin{align*}
f = \text{open( 'fdmdiffus1d_c.dat', 'r' )} \\
\quad \text{tmp = [float(x) for x in f.readline().split( )]} \\
\quad n = \text{int(tmp[0]); alpha = tmp[1]; l = tmp[2]; gamma = tmp[3]}
\end{align*}
\]

will accomplish those tasks. Then, using the information supplied in the first line, we use the statements

\[
\begin{align*}
\text{import numpy as np} \\
\text{import matplotlib.pyplot as plt} \\
x = \text{np.linspace( 0.0,10.0,n+1)}
\end{align*}
\]

to import needed modules and calculate the values of the independent variable at which nodes were placed along the string, and we use the statements

\[
\begin{align*}
t = \text{float( f.readline( ) )} \\
u = \text{[float(tmp) for tmp in f.readline().split( )]} \\
\text{plt.plot( x,u, color='black', linewidth=3 )} \\
\text{plt.title( 't = ' +str(t) )} \\
\text{plt.ylim( [0.0,2.0] )} \\
\text{plt.show()}
\end{align*}
\]

to read the first time and the solution at that first time from the file and generate a quick graph of the initial shape of the string. The resulting graph is shown in the upper left frame of Fig. 15.27.

We could create an animated picture of the evolution of the temperature in this rod in time by continuing to read from the file, displaying each solution in turn. We restore the file pointer to the beginning of the file, reopen the file, and reread the initial line in the file, and recreate the coordinates along the x axis with the statements
Figure 15.27: Temperatures in a rod at various times as it approaches equilibrium. Note that the vertical scale after the graph for $t = 0$ s has been expanded the better to show the gradual evolution of the temperature distribution. This graph was produced with PYTHON from a data file generated by a C program.

Then, we can code a loop to generate the desired animation with the statements

```python
f.close()
f = open( 'fdmdiffus1d_f.dat', 'r' )
tmp = [float(x) for x in f.readline().split()]
n = int(tmp[0]); alpha = tmp[1]; l = tmp[2]; gamma = tmp[3]
x = np.linspace( 0.0,1.0,n+1)

for tp in f:
    t = float(tp) # Convert tp to numerical value
    # Read next solution
    u = [float(tmp) for tmp in f.readline().split()]
    plt.clf() # Clear figure window
    plt.plot( x,u, color='black', linewidth=3 ) # Plot next solution
    plt.title( 't = '+str(t) ) # Add title
    plt.ylim( [0.0,2.0] ) # Standardize y axis range
    plt.pause( 0.1 ) # Display graph
f.close()
plt.show()
```
When run interactively, all of the usual reports of progress in the plotting will be displayed on the screen. If the program is stored in a file and run as a command file, all of that dialog will be suppressed. The resulting output in either case agrees with that displayed in Fig. 15.27.

15.22.3 The Laplace Equation

15.22.3.1 Creating and Testing the C Coding

Discussion of a simple C program to solve the Laplace equation served in Section 9.6 as the vehicle for introducing C as a language. The program there developed, however, was quite crude. In this section we refine that program in several ways, in particular by using a more refined grid and by writing the output to a file to facilitate importing the output into a graphical visualization program. We again address the example laid out in Section 15.16.3, for the solution of which we must set up and solve Eq. (15.174).

The first segment of the program we wish to write will, then, request input of the two parameters, specifically the number of segments into which each side of the domain is to be divided and the length of each side. The statements

```
#include <stdio.h> /* For I/O */
#include <math.h> /* For math functions */
float l, dx; /* For parameters */
int n; /* For parameters */

printf( "Number of segments: "); scanf ( "%d", &n );
printf( "Length of side: "); scanf ( "%f", &l );
dx = l/n;
```

achieve that objective and, in addition, include needed library modules and calculate \( dx \) (which is in our example equal to \( dy \)).

We also need to declare and initialize one-dimensional vectors for the coordinates of points on each axis and a two-dimensional array for the solution. Because dynamic allocation of memory for two-dimensional arrays in C is an advanced topic, we elect instead to make these variables large enough to accommodate the largest size we might imagine, knowing that we can in the present context limit the range of loops to extend over only the portion of the arrays needed for the specified number of segments. To afford some flexibility, however, we define the chosen maximum size as a parameter so, should we need to change it, we need change it in only one place in the coding. Thus, we add the statements

```
#define ln 101 /* Use parameter for maximum dimension */
/* to facilitate editing */
float x[ln], y[ln], u[ln][ln]; /* For solutions */
int i, j; /* For loop counter */

for(i=0; i<=n; i++)
{ x[i]=dx*i; y[i]=dx*i ;
  for(j=0; j<=n; j++)
    u[i][j] = 0.0
}
```

to declare these variables, declare indices for an anticipated pair of nested loops, and initialize the three dimensioned arrays.

Next, we set the Dirichlet boundary values on the left and top edges, and display the initial values in the array \( u \) with the statements

```
for(i=0; i<=n; i++)
{ u[i][n] = 100.0; u[0][i] = 100.0*x[i]/l; }

printf(" Iterate 0 ");
for(i=0; i<=n; i++)
{ for(j=0; j<=n; j++)
   printf( "%.4f", u[i][j] );
   printf( "\n" );
}

In constructing these statements, we have arranged the values in the array \( u \) and the printf statements so that, in the output on the screen the left, right, top, and bottom edges correspond to the left, right, top, and bottom edges in Fig. 15.19. Thus, for example, execution of the statements introduced to this point with \( n \) set to 5 and \( l \) set to 10 yields the output

<table>
<thead>
<tr>
<th>Iterate</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>100.0000 100.0000 100.0000 100.0000 100.0000</td>
</tr>
<tr>
<td></td>
<td>80.0000 0.0000 0.0000 0.0000 0.0000 0.0000</td>
</tr>
<tr>
<td></td>
<td>60.0000 0.0000 0.0000 0.0000 0.0000 0.0000</td>
</tr>
<tr>
<td></td>
<td>40.0000 0.0000 0.0000 0.0000 0.0000 0.0000</td>
</tr>
<tr>
<td></td>
<td>20.0000 0.0000 0.0000 0.0000 0.0000 0.0000</td>
</tr>
<tr>
<td></td>
<td>0.0000 0.0000 0.0000 0.0000 0.0000 0.0000</td>
</tr>
</tbody>
</table>

The values in the top and bottom rows and in the left column of this array correctly reflect the Dirichlet conditions to be imposed on the solution. The values in the right column are unknown initially, since we have imposed Neumann (derivative) conditions along that edge. Those values must be determined as the solution unfolds.

Our task from this point is to determine values to assign to the entries not on the top, bottom, or left edges in such a way that Eq. (15.174) is satisfied. This task differs substantially from the task we confronted in solving the wave and diffusion equations in Sections 15.22.1 and 15.22.2. Those previous equations involved a mixture of boundary and initial conditions and, once the process was started, we could step forward in time as far as we pleased. With the Laplace equation, we are dealing with a boundary value problem alone, and conditions are imposed around the entire perimeter of the region of interest. Rather than having boundary conditions on the left and right of the region of interest, initial conditions on the bottom, and no definite boundary on the top, we have instead boundary conditions on all four sides of the region of interest. One strategy for the present problem would be to guess the derivative \( \partial u/\partial x \) at the left edge, treat the problem as an initial value problem in \( x \) by stepping systematically across the above array to the right edge, and hope that we arrived at that edge with values satisfying the boundary condition at that edge. If we missed, we would make another guess for the starting derivative and try again. We conclude that, in contrast to the problems treated in Sections 15.22.1 and 15.22.2, solution of this problem will require an iterative approach.

We have already written Eq. (15.174) to support an iterative approach more suitable than the guess-and-try approach suggested in the previous paragraph.\(^{59}\) This approach entails (1) guessing a solution at each node not constrained by a Dirichlet condition, (2) stepping systematically through those nodes while replacing the value at each node with the value determined by Eq. (15.174), and (3) repeating step (2) until some criterion of convergence has been met.\(^{60}\) For the moment, let us simply carry out a user-specified number of iterations. Thus, we would add to the input statements the coding

\(^{59}\)This iterative approach has already been described—though for slightly different boundary conditions—in Section 9.3.1 and implemented in C in Section 9.6.
\(^{60}\)We elect to work in place in the array that contains our initial guess, so each newly generated value is used in subsequent calculations as soon as it becomes available. Alternatively, we could store the newly generated values in a second array, completing one pass through the array before using any of the newly generated values. As it turns out, the former approach converges rather more rapidly than the latter approach—and also requires less memory.
int maxits; /* Maximum number of iterations */
int itcnt; /* Iteration counter */
printf( "Maximum number of iterations: "); scanf("%d", &maxits);
itcnt = 0;

to obtain the desired number of iterations and initialize an iteration counter. Further, we add the statement

printf("Iteration %4d", itcnt );

just before writing the solution so as to label the output. Finally, we add the multiple loop

for(itcnt=1; itcnt<=maxits; itcnt++)
{  
  for(i=1; i<n; i++)
    {  
     for(j=1; j<n; j++)
       
      u[i][j]= 0.25*(u[i+1][j]+u[i-1][j]+u[i][j+1]+u[i][j-1]);
       u[i][n]= 0.25*(2.0*u[i][n-1]+u[i-1][n]+u[i+1][n]);
    }
  printf(" Iterate %5d", itcnt );
  for(i=0; i<=n; i++)
    {  
     for(j=0; j<=n; j++)
      
      printf("%10.4f", u[i][j] );
    }
  printf( "\n" );
}

to effect the solution and display the iterates along the way. Here, each pass through the outermost loop will effect one iteration in the algorithm described above. A total of maxits iterations will be executed before the loop terminates. Within that outermost loop, the double loop on i and j steps through the “interior” points in the grid, replacing the value at each with the average of the values at its four nearest neighbors. Then, after exiting from the innermost loop, the node along the right edge (except for the two corner nodes) is processed, replacing the value at each with the value dictated by the appropriate member of Eq. (15.174). Finally, the printf statements display the solution after the current iteration is completed. A more fully commented program containing these statements—and a few others to be discussed in a moment—is named fdmlap2d.c, is listed in Appendix 15.D.3, and can be copied from the directory $HEAD/cc.

As a quick test of our as yet incomplete program and as a way to illustrate the manner of approach to a final solution, let us compile, link, and execute the program as it stands with the input

cc -o fdmlap2d.xc fdmlap2d.c
./fdmlap2d.xc

Number of segments:  5
Length of side:  10
Maximum number of iterations:  5

obtaining the output

Iterate  0
100.0000  100.0000  100.0000  100.0000  100.0000  100.0000

Here and in what follows in this section, we illustrate the statements to be used in a UNIX operating system. Your Local Guide will explain the statements needed in your operating system.
The progression in the solution from iterate to iterate is quite clear, though it is also clear from the changes taking place from one iterate to the next that we haven’t gone far enough to achieve convergence.

While the coding developed to this point certainly displays the essence of the algorithm for solving the Laplace equation, the above output also makes clear that we must carry the solution through a larger number of iterates to achieve convergence. That change, however, will result in much more output unless we suppress all but every $f$-th iteration by adding the coding

```c
int nf;
printf( " Display frequency: "); scanf( "%d", &nf );
```

to request the number of iterates whose display is to be suppressed and by modifying the `printf`
statement in the final loops to make the display conditional upon $\text{itcnt}$ being a multiple of $\text{nf}$. In effect, we simply replace the output statements in the program with the conditional statement\(^{62}\)

```c
if( $\text{nf} \times (\text{itcnt}/\text{nf}) == \text{itcnt}$ )
    {
        printf(" Iterate %5d", \text{itcnt} );
        for(i=0; i<=n; i++)
            { for(j=0; j<=n; j++)
                printf(" %10.4f", u[i][j] );
                printf(" \n");
            }
    }
```

(See the full listing in Appendix 15.D.3 for the details.) With this embellishment, we might submit the starting values

```
./fdmlap2d.xc
Number of segments: 5
Length of side: 10
Maximum number of iterations: 80
Display frequency: 20
```

obtaining the output

```
Iterate  0
  100.0000 100.0000 100.0000 100.0000 100.0000 100.0000
  80.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000
  60.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000
  40.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000
  20.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000
     0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000
Iterate  20
  100.0000 100.0000 100.0000 100.0000 100.0000 100.0000
  80.0000  79.8816  79.8018  79.7600  79.7517  79.7702
  60.0000  59.8314  59.7178  59.6582  59.6464  59.6728
     0.0000  0.0000  0.0000  0.0000  0.0000  0.0000
Iterate  40
  100.0000 100.0000 100.0000 100.0000 100.0000 100.0000
  80.0000  79.9993  79.9988  79.9986  79.9985  79.9986
  60.0000  59.9990  59.9983  59.9979  59.9979  59.9980
     0.0000  0.0000  0.0000  0.0000  0.0000  0.0000
Iterate  60
  100.0000 100.0000 100.0000 100.0000 100.0000 100.0000
  80.0000  80.0000  80.0000  80.0000  80.0000  80.0000
  60.0000  60.0000  60.0000  60.0000  60.0000  60.0000
  40.0000  40.0000  40.0000  40.0000  40.0000  40.0000
  20.0000  20.0000  20.0000  20.0000  20.0000  20.0000
     0.0000  0.0000  0.0000  0.0000  0.0000  0.0000
```

\(^{62}\)Since both $\text{itcnt}$ and $\text{nf}$ are integers, the division $(\text{itcnt}/\text{nf})$ will be an integer division. Thus, $\text{nf} \times (\text{itcnt}/\text{nf})$ will be equal to $\text{itcnt}$ only when $\text{itcnt}$ is a multiple of $j$.\]
Iterate    80
100.0000  100.0000  100.0000  100.0000  100.0000  100.0000
 80.0000  80.0000  80.0000  80.0000  80.0000  80.0000
 60.0000  60.0000  60.0000  60.0000  60.0000  60.0000
 40.0000  40.0000  40.0000  40.0000  40.0000  40.0000
 20.0000  20.0000  20.0000  20.0000  20.0000  20.0000
   0.0000   0.0000   0.0000   0.0000   0.0000   0.0000

In this output, convergence to a steady solution is more evident. Indeed, the solution at iteration 80 conforms to exactly what we might expect.

While the coarse grid used for the above solution renders its display easy on the page of a book and, in particular, allows us to reveal the nature of the iterative convergence to the final steady-state solution, that coarse grid also leaves open the possibility that we have found an accurate solution to the discretized equations that is not, however, a particularly good approximation to the continuous solution to the original problem.\footnote{In fact, the solution we have obtained is \textit{exactly} correct, but the problem we have solved is also especially simple. Actually \textit{obtaining} the exact solution with such a coarse grid is much more an accident of the particular problem than a behavior we can expect.} We would prefer to use a more refined grid and to use graphical rather than tabular display of the solution. To achieve those objectives, we need simply specify a larger number of segments, a larger number of iterations, a larger gap between displayed iterates, and the writing of the solutions to a file for transfer to an available visualization program. We also write the number of segments $n$ and the length $l$ of each side to the file for communication to the visualization program. Inserted at the proper points in the program, the statements

```c
FILE *fpotr; /* For file pointer */
fpotr = fopen( "fdmlap2d_c.dat", "w" );
fprintf( fpotr, "%d %f \n", n, l );
fclose( fptr )
```

declare the necessary file pointer, open the file for writing, write the parameters to the file, and close the file. Changing the \texttt{printf} commands that write the iterate number and the array $u$ to \texttt{fprintf} statements completes the conversion on \texttt{fdmlap2d.c} to the a program that directs its output to the specified file. A more fully commented listing containing these modifications is named \texttt{fdmlap2d_file.c}, is listed in Appendix 15.D.4, and can be copied from the directory \$HEAD/cc.

Once all of these changes have been implemented, we generate the file \texttt{fdmlap2d_c.dat} with the statements

```
c -o fdmlap2d_file.xc fdmlap2d_file.c
./fdmlap2d_file.xc
```

\begin{itemize}
\item Number of segments: 20
\item Length of side: 10
\item Maximum number of iterations: 70
\item Display frequency: 10
\end{itemize}

\subsection{Displaying the Solution Graphically with IDL}

To display this solution in IDL, we would first have to open the file and import not only the first line of data containing the iterate but also the first array containing the solution at iterate 0. The statements
IDL> openr, 1, 'fdmlap2d_c.dat'
IDL> readf, 1, n, l & n=fix(n)
IDL> iterate = "" & readf, 1, iterate & tmp = strtrim(iterate,2)
IDL> iterate = strmid(tmp, 0, 7) + strmid(tmp, 10,14)
IDL> u = fltarr(n+1,n+1) & readf, 1, u

will accomplish that task. Then, using the information supplied in the first line, we use the statement

IDL> x = l*findgen(n+1)/float(n) & y = x

to calculate the values of the independent variables at which nodes were placed along the string. Then, a quick graph of the initial shape can be generated with the statement

IDL> surface, reverse(u,2), x, y, $
IDL> thick=3.0, title=iterate, charsize=3

This picture is in complete agreement with the upper left panel in Fig. 15.28.

We could create an animated picture of the evolution of the temperature distribution in time by continuing to read from the file and display the output with the statements

IDL> while not eof(1) do begin $
IDL> iterate = "" & readf, 1, iterate & tmp = strtrim(iterate,2) & $
IDL> iterate = strmid(tmp, 0, 7) + strmid(tmp, 10,14) & $
IDL> u = fltarr(n+1,n+1) & readf, 1, u & $
IDL> surface, reverse(u,2), x, y, $
IDL> thick=3.0, title=iterate, charsize=3 & $
IDL> wait, 0.3 & $
IDL> endwhile
IDL> close, 1

This output also agrees with that displayed in Fig. 15.28.

15.22.3.3 Displaying the Solution Graphically with MATLAB

To display this solution in MATLAB, we would first have to open the file and import not only the first line of data containing the iterate but also the first array containing the solution at iterate 0. The statements

```matlab
id = fopen( 'fdmlap2d_c.dat', 'r' );
n = fscanf( id, '%d', 1 ); l = fscanf( id, '%f', 1 );
ln = fgetl( id ); iterate = fgetl(id);
u = transpose( fscanf( id, '%f', [n+1,n+1] ) );
```

will accomplish that task. Then, using the information supplied in the first line, we use the statement

```
x, y = meshgrid( 0.0 : l/n : 1, 0.0 : l/n : 1 );
```

to calculate the values of the independent variables at which nodes were placed along the string. Then, a quick graph of the initial shape can be generated with the statement

Note that only one end statement (endwhile) is necessary in this context.
Figure 15.28: Approach of temperature distribution to equilibrium. The panels show the initial distribution (upper left), the distribution at iterate 25 (upper right), the distribution at iterate 50 (lower left), and the distribution at iterate 75 (lower right). This graph was produced with IDL from a data file generated by a C program.

```
mesh( x, y, flipud(u), 'EdgeColor', 'black', 'LineWidth', 3 );
title( iterate, 'FontSize', 20 )
```

This picture is in complete agreement with the upper left panel in Fig. 15.29.

We could create an animated picture of the evolution of the temperature distribution in time by continuing to read from the file, displaying each solution in turn with the statements

```
tmp = fgetl(id); iterate = fgetl(id);
while ( ~feof( id ) )
    u = transpose( fscanf( id, '%f', [n+1,n+1] ) );
    mesh( x, y, flipud(u), 'EdgeColor', 'black', 'LineWidth', 3 );
    title( iterate, 'FontSize', 20 );
    pause( 0.1 );
    tmp = fgetl( id ); iterate = fgetl(id);
end
status = fclose( id )
```
Figure 15.29: Approach of temperature distribution to equilibrium. The panels show the initial distribution (upper left), the distribution at iterate 25 (upper right), the distribution at iterate 50 (lower left), and the distribution at iterate 75 (lower right). This graph was produced with MATLAB from a data file generated by a C program.

This output also agrees with that displayed in Fig. 15.29.

15.22.3.4 Displaying the Solution Graphically with OCTAVE

To display this solution in OCTAVE, we would first have to open the file and import not only the first line of data containing the iterate but also the first array containing the solution at iterate 0. The statements

```octave
id = fopen( 'fdmlap2d_c.dat', 'r' );
n = fscanf( id, '%d', 1 ); l = fscanf( id, '%f', 1 );
ln = fgetl( id );
ln = fgetl(id, 10); iterate = fscanf( id, "%d", 1 );
u = transpose( fscanf( id, "%f", [n+1,n+1] ) );
```

will accomplish that task. Then, using the information supplied in the first line, we use the statement

```octave
[x,y] = meshgrid( 0.0 : l/n : l, 0.0 : l/n : l );
```
Figure 15.30: Approach of temperature distribution to equilibrium. The panels show the initial distribution (upper left), the distribution at iterate 25 (upper right), the distribution at iterate 50 (lower left), and the distribution at iterate 75 (lower right). This graph was produced with OCTAVE from a data file generated by a C program.

to calculate the values of the independent variables at which nodes were placed along the string. Then, a quick graph of the initial shape can be generated with the statement

\[
\text{mesh( x, y, flipud(u), 'EdgeColor', 'black', 'LineWidth', 3 );} \\
\text{title( ['Iterate = ' num2str(iterate, '%5d')], 'FontSize', 20 );}
\]

This picture is in complete agreement with the upper left panel in Fig. 15.30.

We could create an animated picture of the evolution of the temperature distribution in time by continuing to read from the file, displaying each solution in turn with the statements

\[
\text{ln = fgetl( id );} \\
\text{ln = fgetl(id, 10); iterate = fscanf( id, "%d", 1 );} \\
\text{while ( !feof( id ) )} \\
\hspace{1em} \text{u = transpose( fscanf( id, ';f', [n+1,n+1] ) );} \\
\hspace{1em} \text{mesh( x, y, flipud(u), 'EdgeColor', 'black', 'LineWidth', 3 );} \\
\hspace{1em} \text{title( ['Iterate = ', num2str(iterate, '%5d')], 'FontSize', 20 );} \\
\hspace{1em} \text{pause( 0.1 );} \\
\hspace{1em} \text{ln = fgetl( id );} \\
\hspace{1em} \text{ln = fgetl(id, 10); iterate = fscanf( id, "%d", 1 );}
\]
CHAPTER 15. SOLVING PARTIAL DIFFERENTIAL EQUATIONS

258

end
status = fclose( id )

This output also agrees with that displayed in Fig. 15.30.

15.22.3.5 Displaying the Solution Graphically with PYTHON

To display this solution in PYTHON, we would first have to open the file, import the first line of data containing the number n of segments in each side of the region and the length l of each side. The statements

```
f = open( 'fdmlap2d_c.dat', 'r' ) # Open file
tmp = [float(x) for x in f.readline().split()] # Read first line
n = int(tmp[0]); l = tmp[1] # Separate components in line
```

will accomplish that task. Then, we read the initial values of u over the region with the statements\(^65\)

```
iterate = f.readline().strip() # Read iterate as string
u = [] # Initialize array for solution
for j in range(n+1): # Read each row in array
    row = [float(tmp) for tmp in f.readline().split()]
u.append(row) # Append row to array
```

After importing needed modules with the statements

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

we use the information supplied in the first line in the statements

```
xx = np.linspace(0.0,l, n+1) # Create grid over which to plot solution
x,y = np.meshgrid(xx,xx)
```

to calculate the values of the independent variables at which nodes were placed along the edges of the region and to create the grid over which the solution will ultimately be plotted. A quick graph of the initial shape can then be generated with the statements

```
u1 = np.flip(u, axis=0) # Flip array to repair y values
ax1 = plt.axes(projection='3d') # Set for 3D display
ax1.plot_surface( x, y, u1, color='white', shade=False, edgecolor='black')
ax1.set_xlabel('x'); ax1.set_ylabel('y'); ax1.set_zlabel('u')
ax1.set_title( iterate )
ax1.view_init(azim=-45, elev=25) # Reorient for best viewing
plt.show() # Display array
f.close() # Close file
```

This picture is in complete agreement with the upper left panel in Fig. 15.31.

\(^65\)We know from the structure of the file that each row in the array being read occupies one line in the file and there are \(n+1 = 21\) rows in each array. Because we can read an entire row in a single statement, we need only a single loop to read the full array.
Figure 15.31: Approach of temperature distribution to equilibrium. The panels show the initial distribution (upper left), the distribution at iterate 50 (upper right), the distribution at iterate 100 (lower left), and the distribution at iterate 150 (lower right). This graph was produced with PYTHON from a data file generated by a C program.

Creating an animated display of the several steps as the iteration proceeds requires a bit more elaborate coding. We begin by importing needed modules, opening the file, reading the first line, and creating the necessary grid with the statements

```python
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

f = open( 'fdmlap2d_c.dat', 'r' ) # Open file
tmp = [float(x) for x in f.readline().split()] # Read first line
n = int(tmp[0]); l = tmp[1] # Separate components in line

xx = np.linspace(0.0,l, n+1) # Create grid over which to plot solution
x,y = np.meshgrid(xx,xx)
```

Next, using the iterate to control the loop, we read each iterate and the corresponding values of u in turn and display the shape with the statements

```python
iterate=f.readline().strip() # Read iterate as a string
```
while( iterate !="" ):
    # Continue while not at end
    u=[]
    for j in range(n+1):
        # Read each row in array
        row = [float(tmp) for tmp in f.readline().split()]
        u.append(row) # Append row to array
    u1 = np.flip(u, axis=0)
    plt.clf() # Clear current plot
    ax1 = plt.axes(projection='3d')
    ax1.plot_surface( x, y, u1, color='white', shade=False, edgecolor='black')
    ax1.set_xlabel('x'); ax1.set_ylabel('y'); ax1.set_zlabel('u')
    ax1.set_title( iterate )
    ax1.view_init(azim=-45, elev=25)
    plt.pause(1.0)
    iterate=f.readline().strip() # Read next iterate
plt.show() # Effect clean exit
f.close() # Close file

This output also agrees with that displayed in Fig. 15.31. carried out here. We therefore elect not to repeat that discussion.

15.23 Finite Element Methods (FEMs) in Two Dimensions

While their implementation is more involved in two dimensions than in one dimension, finite element approaches to problems in two dimensions involve the same several steps as those identified at the beginning of Section 15.9. In this section, we illustrate the method of finite element analysis by applying those steps to the two-dimensional boundary value problem defined in Section 15.1.8.

15.23.1 Discretizing the Domain: Pre-processing

The first step is to divide the domain into elements or, in the jargon, to create a mesh over the domain or to mesh the domain. To facilitate the creation of that mesh (and the discussion of the method), we elect to use the simplest possible two-dimensional element, meshing the region of interest—denote it by $\Omega$—with triangular elements as illustrated, for example, in the left panel of Fig. 15.32. Each element is defined by three nodes (vertices), which also serve as nodes on adjacent elements, and each element has three edges—the lines along which pairs of elements meet. Together, the edges define the perimeter or boundary of the element. For the meshing to be legal, no node in one element can be located along the edge of another element, as, for example, at the point circled in the right panel of Fig. 15.32. Further, since the error in the finite element solution varies inversely as the sine of the smallest angle in each element, all elements should be at least approximately equilateral. While increasing the number of elements in the region improves the accuracy of the solution, it also increases both the computation time and the memory needed to obtain the solution. Thus, the number of elements should be large (and their size small) only in regions in which the solution is expected to vary rapidly. A coarser mesh can be accepted in regions in which the solution varies slowly. Choosing an appropriate mesh of elements of suitable sizes is difficult and may require an iterative approach, especially if little is known a priori about the solution.

As a modest example, suppose the region of interest is a rectangle. For simplicity, divide that region into eight triangular elements by one vertical line, one horizontal line, and three diagonal lines, as shown in Fig. 15.33. Then, we number the elements in an order that seems convenient. In addition, we number the nodes both globally (i.e., with respect to the entire domain) and locally (i.e., with respect to a particular element). The global labeling is done by assigning a specific and
Figure 15.32: Discretization of a general region with triangular elements.

Figure 15.33: Subdivision of a rectangular domain into eight triangular elements.

Figure 15.34: Rectangular domain of Fig. 15.33 with global (left) and local (right) node numbers.
Table 15.2: The connectivity matrix for the example in Section 15.23.

<table>
<thead>
<tr>
<th>Element</th>
<th>Global number of local node 0</th>
<th>Global number of local node 1</th>
<th>Global number of local node 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>8</td>
<td>7</td>
</tr>
</tbody>
</table>

unique integer to each node, as in the left panel of Fig. 15.34. Locally, the nodes—remember that each element has three nodes—are numbered with the integers 0, 1, and 2, as in the right panel of Fig. 15.34. Each element will have its own zeroth, first, and second nodes, each of which also has a global label that identifies it uniquely. A connectivity matrix such as the one shown in Table 15.2 is created to express the relation between global and local node numbers. In the present context, the matrix consists of \( M \) rows, one for each element, with each row containing 3—one for each node of the element—entries, i.e. the matrix has \( M \) rows and three columns. Every entry in the matrix is a global node number corresponding to the zeroth, first, or second node of each element. In this example, the (global) numbering both of the elements and of the nodes is done in the same way, from top to bottom and from left to right. The local node numbers are assigned counterclockwise starting at the lower left node of each element. For element 0, for example, the global number of local node 0 is 1, the global number of local node 1 is 3, and the global number of local node 2 is 0. Similarly, for element 1, we have global node numbers 1, 4, and 3. The rest of the entries are generated in the same way and are recorded in Table 15.2.

15.23.2 Selecting the Interpolation or Shape Functions

The simplest interpolation or shape functions are linear in both \( x \) and \( y \). For element \( e \), we write the approximating equation in the form

\[
\tilde{\varphi}^{(e)}(x, y) = a^{(e)} + b^{(e)} x + c^{(e)} y
\] (15.178)

The constants are determined so that this equation gives the correct values at all three nodes of any element, i.e., so that

\[
\begin{align*}
\tilde{\varphi}_0^{(e)} &= a^{(e)} + b^{(e)} x_0^{(e)} + c^{(e)} y_0^{(e)} \\
\tilde{\varphi}_1^{(e)} &= a^{(e)} + b^{(e)} x_1^{(e)} + c^{(e)} y_1^{(e)} \\
\tilde{\varphi}_2^{(e)} &= a^{(e)} + b^{(e)} x_2^{(e)} + c^{(e)} y_2^{(e)}
\end{align*}
\] (15.179)

where subscripts refer to local node numbers. Solution of this system of equations for \( a^{(e)} \), \( b^{(e)} \), and \( c^{(e)} \) and substitution of the solution into Eq. (15.178) gives the approximating function

\[
\tilde{\varphi}^{(e)}(x, y) = \sum_{j=0}^{2} N_j^{(e)}(x, y) \tilde{\varphi}_j^{(e)}
\] (15.180)

\[\text{In this example, we elect to begin all numbering of elements and nodes—both global and local—with zero, fully aware that some programming languages (IDL, PYTHON, C) follow that pattern but that others (MATLAB, OCTAVE, FORTRAN) start numbering with 1. In the latter case, each index in the program will be incremented by 1 relative to those in this theoretical discussion of the general strategies.}\]

\[\text{See Exercise 15.15.}\]
where, with $\Delta^{(e)}$—the determinant of the matrix of coefficients in Eq. (15.179)—defined by

$$
\Delta^{(e)} = \left| \begin{array}{ccc}
1 & x_0^{(e)} & y_0^{(e)} \\
1 & x_1^{(e)} & y_1^{(e)} \\
1 & x_2^{(e)} & y_2^{(e)}
\end{array} \right| \quad (15.181)
$$

the interpolation or shape functions $N_j^{(e)}(x,y)$ are given by

$$
N_0^{(e)}(x,y) = \frac{1}{\Delta^{(e)}} \left| \begin{array}{ccc}
1 & x & y \\
1 & x_0^{(e)} & y_0^{(e)} \\
1 & x_2^{(e)} & y_2^{(e)}
\end{array} \right| \quad (15.182)
$$

$$
N_1^{(e)}(x,y) = \frac{1}{\Delta^{(e)}} \left| \begin{array}{ccc}
1 & x_0^{(e)} & y_0^{(e)} \\
1 & x & y \\
1 & x_2^{(e)} & y_2^{(e)}
\end{array} \right| \quad (15.183)
$$

$$
N_2^{(e)}(x,y) = \frac{1}{\Delta^{(e)}} \left| \begin{array}{ccc}
1 & x_0^{(e)} & y_0^{(e)} \\
1 & x_1^{(e)} & y_1^{(e)} \\
1 & x & y
\end{array} \right| \quad (15.184)
$$

From these expressions and the properties of determinants, we can quickly recognize that each interpolation function has the value one at the node corresponding to its index and the value zero at the other two nodes. Thus, the sum on the right side of Eq. (15.180) reduces at each node to the value given on the left side. Equally true but less obvious, each interpolation function is in fact zero along the entire line joining the two nodes at which it has the value zero. This property will turn out to be useful when the boundary conditions are taken into account.

### 15.23.3 Formulating the Equations for a Single Element

From the general form of Eq. (15.70), the expression for the residual $r$ can be written easily as

$$
r = -\frac{\partial}{\partial x} \left( \alpha_x \frac{\partial \tilde{\phi}}{\partial x} \right) - \frac{\partial}{\partial y} \left( \alpha_y \frac{\partial \tilde{\phi}}{\partial y} \right) + \beta \tilde{\phi} - f \quad (15.185)
$$

Evaluated for element $e$, the weighted residuals then are given by the integral

$$
R_i^{(e)} = \int \int_{\Omega^{(e)}} N_i^{(e)} r \, dx \, dy ; \quad i = 0, 1, 2 \quad (15.186)
$$

over the surface $\Omega^{(e)}$ of element $e$. Here, following the Galerkin formulation, we take the weighting functions to be the interpolation functions themselves. Substitution of Eq. (15.185) into Eq. (15.186) leads to

$$
R_i^{(e)} = \int \int_{\Omega^{(e)}} N_i^{(e)} \left[ -\frac{\partial}{\partial x} \left( \alpha_x \frac{\partial \tilde{\phi}}{\partial x} \right) - \frac{\partial}{\partial y} \left( \alpha_y \frac{\partial \tilde{\phi}}{\partial y} \right) + \beta \tilde{\phi} - f \right] \, dx \, dy \quad (15.187)
$$

$$
= -\int \int_{\Omega^{(e)}} N_i^{(e)} \left( \frac{\partial}{\partial x} \left( \alpha_x \frac{\partial \tilde{\phi}}{\partial x} \right) + \frac{\partial}{\partial y} \left( \alpha_y \frac{\partial \tilde{\phi}}{\partial y} \right) \right) \, dx \, dy
$$

$$
+ \int \int_{\Omega^{(e)}} N_i^{(e)} \beta \tilde{\phi} \, dx \, dy - \int \int_{\Omega^{(e)}} N_i^{(e)} f \, dx \, dy \quad (15.188)
$$

---

68See Exercise 15.15.
Then, interpreting $\psi$, $V_x$, and $V_y$ in Eq. (15.232), the identity developed in Appendix 15.E, as $N_i^{(e)}$, $\alpha_x \partial \tilde{\phi}/\partial x$, and $\alpha_y \partial \tilde{\phi}/\partial y$, respectively, we can rewrite the first integral in Eq. (15.188) so as to recast the entirety of Eq. (15.188) in the form

$$R_i^{(e)} = \int \int_{\Omega^{(e)}} \left( \alpha_x \frac{\partial N_i^{(e)}}{\partial x} \frac{\partial \tilde{\phi}}{\partial x} + \alpha_y \frac{\partial N_i^{(e)}}{\partial y} \frac{\partial \tilde{\phi}}{\partial y} + \beta N_i^{(e)} \tilde{\phi}^{(e)} \right) dx dy$$

$$- \int \int_{\Omega^{(e)}} N_i^{(e)} f dx dy - \oint_{\Gamma^{(e)}} N_i^{(e)} \left( \alpha_x \frac{\partial \tilde{\phi}}{\partial x} \hat{i} + \alpha_y \frac{\partial \tilde{\phi}}{\partial y} \hat{j} \right) \cdot \hat{n}^{(e)} dl$$  \hspace{1cm} (15.189)

Here, $\Gamma^{(e)}$ is the path bounding the region $\Omega^{(e)}$, and $\hat{n}^{(e)}$ is a unit vector lying in the plane of element $e$, perpendicular at each point to $\Gamma^{(e)}$, and directed outward from the perspective of a viewer in $\Omega^{(e)}$. Substitution of the approximating function of Eq. (15.180) yields for the weighted residuals the expression

$$R_i^{(e)} = 2 \sum_{j=0}^{2} \phi_j^{(e)} \int \int_{\Omega^{(e)}} \left( \alpha_x \frac{\partial N_i^{(e)}}{\partial x} \frac{\partial N_j^{(e)}}{\partial x} + \alpha_y \frac{\partial N_i^{(e)}}{\partial y} \frac{\partial N_j^{(e)}}{\partial y} + \beta N_i^{(e)} N_j^{(e)} \right) dx dy$$

$$- \int \int_{\Omega^{(e)}} N_i^{(e)} f dx dy - \oint_{\Gamma^{(e)}} N_i^{(e)} \left( \alpha_x \frac{\partial \tilde{\phi}}{\partial x} \hat{i} + \alpha_y \frac{\partial \tilde{\phi}}{\partial y} \hat{j} \right) \cdot \hat{n}^{(e)} dl$$ \hspace{1cm} (15.190)

or, in the matrix form,

$$\{R^{(e)}\} = [K^{(e)}] \{\tilde{\phi}^{(e)}\} - \{b^{(e)}\} - \{g^{(e)}\}$$  \hspace{1cm} (15.191)

where $[\ldots]$ denotes a matrix and $\{\ldots\}$ denotes a vector. The matrix $[K^{(e)}]$ is a $3 \times 3$ matrix and $\{R^{(e)}\}$, $\{\tilde{\phi}^{(e)}\}$, $\{b^{(e)}\}$, and $\{g^{(e)}\}$ are three-element vectors. The matrix elements $K_{ij}^{(e)}$ and the vector elements $b_i^{(e)}$ and $g_i^{(e)}$ can be obtained from Eq. (15.190), namely

$$K_{ij}^{(e)} = \int \int_{\Omega^{(e)}} \left( \alpha_x \frac{\partial N_i^{(e)}}{\partial x} \frac{\partial N_j^{(e)}}{\partial x} + \alpha_y \frac{\partial N_i^{(e)}}{\partial y} \frac{\partial N_j^{(e)}}{\partial y} + \beta N_i^{(e)} N_j^{(e)} \right) dx dy$$ \hspace{1cm} (15.192)

$$b_i^{(e)} = \int \int_{\Omega^{(e)}} N_i^{(e)} f dx dy$$ \hspace{1cm} (15.193)

$$g_i^{(e)} = \oint_{\Gamma^{(e)}} N_i^{(e)} \left( \alpha_x \frac{\partial \tilde{\phi}}{\partial x} \hat{i} + \alpha_y \frac{\partial \tilde{\phi}}{\partial y} \hat{j} \right) \cdot \hat{n}^{(e)} dl$$ \hspace{1cm} (15.194)

where, in all cases, $i$ and $j$ assume the values 0, 1, and 2.

All elements of the matrix $[K^{(e)}]$ and of the vector $\{b^{(e)}\}$ are known at the start of the problem. The elements of the vector $\{g^{(e)}\}$ are ultimately determined when the boundary conditions are incorporated. Those terms will be discussed further in Section 15.23.5.

Of course, the equations determining the unknown nodal values are obtained by requiring the weighted residuals to be zero. In matrix form, we have from Eq. (15.191) for the $e$-th element that

$$[K^{(e)}] \{\tilde{\phi}^{(e)}\} = \{b^{(e)}\} + \{g^{(e)}\}$$ \hspace{1cm} (15.195)

Written out in terms of the specific (local) nodes, this equation becomes

$$\begin{pmatrix}
R_{00}^{(e)} & R_{01}^{(e)} & R_{02}^{(e)} \\
R_{10}^{(e)} & R_{11}^{(e)} & R_{12}^{(e)} \\
R_{20}^{(e)} & R_{21}^{(e)} & R_{22}^{(e)}
\end{pmatrix}
\begin{pmatrix}
\tilde{\phi}_0^{(e)} \\
\tilde{\phi}_1^{(e)} \\
\tilde{\phi}_2^{(e)}
\end{pmatrix}
= \begin{pmatrix}
b_0^{(e)} \\
b_1^{(e)} \\
b_2^{(e)}
\end{pmatrix}
+ \begin{pmatrix}
g_0^{(e)} \\
g_1^{(e)} \\
g_2^{(e)}
\end{pmatrix}$$ \hspace{1cm} (15.196)
15.23.4 Assembling the System of Equations

The goal of assembly is to generate a system of equations that can be solved for the vector \( \{ \hat{\varphi} \} \), whose—here 9—components approximate the solution to the original problem at the nodes. That is, we seek a set of equations of the form

\[
[K]\{ \hat{\varphi} \} = \{ b \} + \{ g \}
\]  

(15.197)

where \([K]\) is a—here 9 x 9—matrix and \( \{ b \} \) and \( \{ g \} \) are—here 9 component—vectors. As in the one-dimensional case discussed in Section 15.9, the equations for the individual elements are assembled to take into account the requirement that the solution be continuous at the boundaries between elements. Regardless of which element one focuses on, the solution along each bounding line is a linear interpolation between the values at the two nodes defining the line. Thus, requiring that the solution be continuous at each node assures continuity along the lines joining all nodes.

To illustrate the process, consider the assembly of the equations relating to elements 0 and 1. As shown in Fig. 15.34 and in the connectivity matrix, the global numbers of the three nodes of element 0 are (in order) 1, 3, and 0. Thus,

\[
\phi_0^{(0)} = \hat{\varphi}_1; \quad \phi_1^{(0)} = \hat{\varphi}_3; \quad \phi_2^{(0)} = \hat{\varphi}_0
\]  

(15.198)

and, written with global identifications on the unknown \( \hat{\varphi} \)'s, Eq. (15.196) for element 0 becomes

\[
\begin{bmatrix}
K_{00}^{(0)} & K_{01}^{(0)} & K_{02}^{(0)} \\
K_{10}^{(0)} & K_{11}^{(0)} & K_{12}^{(0)} \\
K_{20}^{(0)} & K_{21}^{(0)} & K_{22}^{(0)}
\end{bmatrix}
\begin{bmatrix}
\hat{\varphi}_1 \\
\hat{\varphi}_3 \\
\hat{\varphi}_0
\end{bmatrix} = 
\begin{bmatrix}
b_0^{(0)} \\
b_1^{(0)} \\
b_2^{(0)}
\end{bmatrix} + 
\begin{bmatrix}
g_0^{(0)} \\
g_1^{(0)} \\
g_2^{(0)}
\end{bmatrix}
\]  

(15.199)

Similarly, the three nodes of element 1 are (in order) 1, 4, and 3. Thus,

\[
\phi_0^{(1)} = \hat{\varphi}_1; \quad \phi_1^{(1)} = \hat{\varphi}_4; \quad \phi_2^{(1)} = \hat{\varphi}_3
\]  

(15.200)

and Eq. (15.196) for element 1 becomes

\[
\begin{bmatrix}
K_{00}^{(1)} & K_{01}^{(1)} & K_{02}^{(1)} \\
K_{10}^{(1)} & K_{11}^{(1)} & K_{12}^{(1)} \\
K_{20}^{(1)} & K_{21}^{(1)} & K_{22}^{(1)}
\end{bmatrix}
\begin{bmatrix}
\hat{\varphi}_1 \\
\hat{\varphi}_3 \\
\hat{\varphi}_0
\end{bmatrix} = 
\begin{bmatrix}
b_0^{(1)} \\
b_1^{(1)} \\
b_2^{(1)}
\end{bmatrix} + 
\begin{bmatrix}
g_0^{(1)} \\
g_1^{(1)} \\
g_2^{(1)}
\end{bmatrix}
\]  

(15.201)

We have, of course, found six equations constraining only four unknowns, namely \( \hat{\varphi}_0, \hat{\varphi}_1, \hat{\varphi}_3, \) and \( \hat{\varphi}_4 \). Two of the six are redundant. Rather than discard two of them, however, we elect to reduce the number to four by adding selected pairs of these equations. To see the proper combinations, let us recast each of Eqs. (15.199) and (15.201) as equations for the vector containing all four of the involved nodal values. That is, let us augment each equation to make it more obvious that it is not the only equation with which we must deal. In the process, we add, for example, to Eq. (15.199) a row and a column corresponding to global node 4 even though that node does not enter into the equations for element 0. Further, we rearrange the order of the columns in the matrix of Eq. (15.199) so the vector of unknowns can be written with the entries in the order of the global node numbers and we rearrange the order of the rows so that the entries in \( \{ b \} \) and \( \{ g \} \) are also in the order of the global node numbers (and, incidentally and automatically, so that the symmetry of the augmented matrix is preserved). The result is

\[
\begin{bmatrix}
K_{22}^{(0)} & K_{20}^{(0)} & K_{21}^{(0)} & 0 \\
K_{02}^{(0)} & K_{00}^{(0)} & K_{01}^{(0)} & 0 \\
K_{12}^{(0)} & K_{10}^{(0)} & K_{11}^{(0)} & 0 \\
0 & 0 & 0 & 0
\end{bmatrix}
\begin{bmatrix}
\hat{\varphi}_0 \\
\hat{\varphi}_1 \\
\hat{\varphi}_3 \\
\hat{\varphi}_4
\end{bmatrix} = 
\begin{bmatrix}
b_0^{(0)} \\
b_1^{(0)} \\
b_2^{(0)} \\
0
\end{bmatrix} + 
\begin{bmatrix}
g_0^{(0)} \\
g_1^{(0)} \\
g_2^{(0)} \\
0
\end{bmatrix}
\]  

(15.202)

\(^{69}\)We assume that the equations are not contradictory.
Similarly, we augment Eq. (15.201) to obtain the result

\[
\begin{bmatrix}
0 & 0 & 0 & 0 \\
0 & K_{00}^{(1)} & K_{02}^{(1)} & K_{01}^{(1)} \\
0 & K_{20}^{(1)} & K_{22}^{(1)} & K_{21}^{(1)} \\
0 & K_{10}^{(1)} & K_{12}^{(1)} & K_{11}^{(1)}
\end{bmatrix}
\begin{bmatrix}
\tilde{\phi}_0 \\
\tilde{\phi}_1 \\
\tilde{\phi}_3 \\
\tilde{\phi}_4
\end{bmatrix}
= \begin{bmatrix}
0 \\
b_0^{(1)} \\
b_2^{(1)} \\
b_4^{(1)}
\end{bmatrix}
+ \begin{bmatrix}
g_0^{(1)} \\
g_2^{(1)} \\
g_4^{(1)}
\end{bmatrix}
\tag{15.203}
\]

Basically, we reduce the original set of six equations to the required four by adding these equations, finding as the result of assembling the equations for elements 0 and 1 that

\[
\begin{bmatrix}
K_{22}^{(0)} & K_{20}^{(0)} & K_{21}^{(0)} & 0 \\
K_{02}^{(0)} & K_{00}^{(0)} + K_{00}^{(1)} & K_{01}^{(0)} + K_{01}^{(1)} & K_{02}^{(1)} \\
K_{12}^{(0)} & K_{10}^{(0)} + K_{10}^{(1)} & K_{11}^{(0)} + K_{11}^{(1)} & K_{12}^{(1)} \\
0 & K_{10}^{(1)} & K_{12}^{(1)} & K_{11}^{(1)}
\end{bmatrix}
\begin{bmatrix}
\tilde{\phi}_0 \\
\tilde{\phi}_1 \\
\tilde{\phi}_3 \\
\tilde{\phi}_4
\end{bmatrix}
= \begin{bmatrix}
b_2^{(0)} \\
b_0^{(0)} + b_0^{(1)} \\
b_1^{(0)} + b_2^{(1)} \\
b_1^{(1)}
\end{bmatrix}
+ \begin{bmatrix}
g_2^{(0)} \\
g_0^{(0)} + g_0^{(1)} \\
g_1^{(0)} + g_2^{(1)} \\
g_1^{(1)}
\end{bmatrix}
\tag{15.204}
\]

Furthermore, we can expand these equations to the full 9 × 9 set for all of the nodes, finding that, at this stage in the full assembly,

\[
\begin{bmatrix}
K_{22}^{(0)} & K_{20}^{(0)} & 0 & K_{21}^{(0)} & 0 & 0 & 0 & 0 & 0 \\
K_{02}^{(0)} & K_{00}^{(0)} + K_{00}^{(1)} & 0 & K_{01}^{(0)} + K_{01}^{(1)} & K_{02}^{(1)} & 0 & 0 & 0 & 0 \\
K_{12}^{(0)} & K_{10}^{(0)} + K_{10}^{(1)} & 0 & K_{11}^{(0)} + K_{11}^{(1)} & K_{12}^{(1)} & 0 & 0 & 0 & 0 \\
0 & K_{10}^{(1)} & 0 & K_{12}^{(1)} & K_{11}^{(1)} & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
\begin{bmatrix}
\tilde{\phi}_0 \\
\tilde{\phi}_1 \\
\tilde{\phi}_3 \\
\tilde{\phi}_4 \\
\tilde{\phi}_5 \\
\tilde{\phi}_6 \\
\tilde{\phi}_7 \\
\tilde{\phi}_8
\end{bmatrix}
= \begin{bmatrix}
b_2^{(0)} \\
b_0^{(0)} + b_0^{(1)} \\
b_1^{(0)} + b_2^{(1)} \\
b_1^{(1)}
\end{bmatrix}
+ \begin{bmatrix}
g_2^{(0)} \\
g_0^{(0)} + g_0^{(1)} \\
g_1^{(0)} + g_1^{(1)} \\
g_1^{(1)}
\end{bmatrix}
\tag{15.205}
\]

The pattern is now clear:

- The element \(K_{ij}^{(e)}\) of the 3 × 3 matrix \([K^{(e)}]\) contributes additively to the element \(K_{lm}\) of the 9 × 9 assembled matrix \([K]\), where \(l\) is the global number of the node with local number \(i\)
in element $e$ and $m$ is the global number of the node with local number $j$ in element $e$. For example, $K_{00}^{(1)}$ contributes to $K_{11}$ because local node 0 in element 0 coincides with global node 1; $K_{10}^{(1)}$ contributes to $K_{41}$ because local node 1 of element 1 coincides with global node 4 and local node 0 of element 1 coincides with global node 1.

- The element $b_i^{(e)}$ contributes additively to the element $b_l$, where $l$ is the global number of the node with local number $i$ in element $e$. For example, $b_1^{(0)}$ contributes to $b_3$ because local node 0 in element 0 coincides with global node 1.

- The element $g_i^{(e)}$ contributes additively to the element $g_l$, where $l$ is the global number of the node with local number $i$ in element $e$. For example, $g_2^{(1)}$ contributes to $g_3$ because local node 2 of element 1 coincides with global node 3.

From this point, continuing the assembly to include the contributions from the remaining elements is straightforward but tedious.

15.23.5 Incorporating the Boundary Conditions

Apart from the nodal values of $\tilde{\varphi}$, only the $g$’s in Eq. (15.205)—or better in what Eq. (15.205) becomes when all elements have been incorporated—remain unknown. To demonstrate how the boundary conditions either render a priori knowledge of these quantities unnecessary or specify these quantities explicitly, we begin by noting that the fully assembled vector $\{g\}$ is

$$\{g\} = \begin{bmatrix}
g_2^{(0)} \\
g_0^{(0)} + g_0^{(1)} + g_2^{(2)} \\
g_0^{(2)} + g_0^{(3)} \\
g_1^{(0)} + g_2^{(1)} + g_2^{(4)} \\
g_1^{(1)} + g_1^{(2)} + g_2^{(3)} + g_0^{(4)} + g_0^{(5)} + g_2^{(6)} \\
g_1^{(3)} + g_0^{(6)} + g_0^{(7)} \\
g_1^{(4)} + g_2^{(5)} \\
g_1^{(5)} + g_1^{(6)} + g_2^{(7)} \\
g_1^{(7)}
\end{bmatrix}$$  \hspace{1cm} (15.206)

Remember, now, that $g_i^{(e)}$ is given by the line integral

$$g_i^{(e)} = \oint_{\Gamma^{(e)}} N_i^{(e)} \left( \alpha_x \frac{\partial \tilde{\varphi}^{(e)}}{\partial x} \hat{i} + \alpha_y \frac{\partial \tilde{\varphi}^{(e)}}{\partial y} \hat{j} \right) \cdot \hat{n}^{(e)} \, dl$$  \hspace{1cm} (15.207)

[see Eq. (15.194)] evaluated counterclockwise around the entirety of element $e$. Furthermore, notice that the various $g$’s that are added together in each row of Eq. (15.206) are the line integrals around each element that has a vertex at the global node with which the row in $\{g\}$ is associated.

Let us look at a few representative elements of $\{g\}$. Suppose we represent the line integrals by a notation like $(i \rightarrow j + j \rightarrow k + k \rightarrow i)$ for the integral that runs from (global) node $i$ to (global) node $j$ to (global) node $k$ and back to (global) node $i$ around a particular element. Because the integrand must be continuous along and across the lines separating elements, we need not actually keep track of which element we are traversing on any of the indicated segments. In this notation,
for example, \( g_4 \), which is the sum of the line integrals around all the elements having a vertex at global node number 4, becomes

\[
(4 \rightarrow 3 + 3 \rightarrow 1 + 1 \rightarrow 4) +
(4 \rightarrow 1 + 1 \rightarrow 2 + 2 \rightarrow 4) +
(4 \rightarrow 2 + 2 \rightarrow 5 + 5 \rightarrow 4) +
(4 \rightarrow 5 + 5 \rightarrow 7 + 7 \rightarrow 4) +
(4 \rightarrow 7 + 7 \rightarrow 6 + 6 \rightarrow 4) +
(4 \rightarrow 6 + 6 \rightarrow 3 + 3 \rightarrow 4)
\]

Next, note that the middle integral in each line is zero, because the interpolation function along that line is identically zero at all points.\(^{70}\) Then, note that each of the other integrals appears twice, traversed once in each direction, and the two together cancel. This particular sum boils down to zero! The same behavior characterizes all nodes not on the boundary of \( \Omega \). (There is only one such node—global node 4—in the present example.)

We can examine other elements in the vector \( \{g\} \) using the same shorthand notation. For example, the element \( g_2 \) would be represented by

\[
(2 \rightarrow 4 + 4 \rightarrow 1 + 1 \rightarrow 2) +
(2 \rightarrow 5 + 5 \rightarrow 4 + 4 \rightarrow 2)
\]

As in the previous paragraph, the middle integral in each item is zero and the first integral in the first item cancels the third integral in the second item. Since this node is on the boundary, however, complete cancellation does not happen; we are left with \( 1 \rightarrow 2 + 2 \rightarrow 5 \). Similar considerations reduce the vector \( \{g\} \) to the much simpler vector

\[
\begin{align*}
\{g\} &= \{3 \rightarrow 0 + 0 \rightarrow 1 \\
&\quad 0 \rightarrow 1 + 1 \rightarrow 2 \\
&\quad 1 \rightarrow 2 + 2 \rightarrow 5 \\
&\quad 6 \rightarrow 3 + 3 \rightarrow 0 \\
&\quad 0 \\
&\quad 2 \rightarrow 5 + 5 \rightarrow 8 \\
&\quad 7 \rightarrow 6 + 6 \rightarrow 3 \\
&\quad 8 \rightarrow 7 + 7 \rightarrow 6 \\
&\quad 5 \rightarrow 8 + 8 \rightarrow 7
\end{align*}
\]  \hspace{1cm} (15.208)
\]

Only integrals along edges that are on the boundary of \( \Omega \) remain. Resolution of all remaining unknowns therefore lies in the boundary conditions!

The explicit treatment of these terms from this point depends on what type of boundary conditions are specified. With Dirichlet conditions, the solution is specified on the boundary and the value at each node so constrained is therefore fixed by the boundary conditions. To illustrate the incorporation of this boundary condition, consider a Dirichlet condition at a particular node \( m \). We first replace the row corresponding to global node \( m \) in the equation \( [K]\{\tilde{\phi}\} = \{b + g\} \) with the equation \( \tilde{\phi}_m = p_m \), where \( p_m \) is the value specified for node \( m \) by the boundary condition. In short,

- The element in the \( m \)-th column of the \( m \)-th row of \( [K] \) is replaced by 1,
- All other elements in that row are replaced by 0, and
- The \( m \)-th element in the vector \( \{b + g\} \) is replaced by \( p_m \).\(^{71}\)

Then, to complete the incorporation of this boundary condition, we must reflect the influence of this known value \( \tilde{\phi}_m \) on the remaining equations in order to restore the symmetry of the stiffness matrix. As an example, consider the \( n \)-th equation \( (n \neq m) \) in the system:

\[
K_{n0} \tilde{\phi}_0 + K_{n1} \tilde{\phi}_1 + \cdots + K_{nm} \tilde{\phi}_m + \cdots + K_{n7} \tilde{\phi}_7 + K_{n8} \tilde{\phi}_8 = b_n + g_n \hspace{1cm} (15.209)
\]

\(^{70}\)See Exercise 15.15.

\(^{71}\)The original \( m \)-th equation is then interpreted as an equation giving \( g_m \) after the solution to the altered set of equations has been found.
Since \( \tilde{\varphi}_m = p_m \), this expression can be recast as

\[
K_{n0} \tilde{\varphi}_0 + K_{n1} \tilde{\varphi}_1 + \cdots + 0 \tilde{\varphi}_m + \cdots + K_{n7} \tilde{\varphi}_7 + K_{n8} \tilde{\varphi}_8 = b_n + g_n - K_{nm} p_m
\]  

(15.210)

Such recasting of all equations for \( n \neq m \) is accomplished by

- Multiplying by \( p_m \) all elements except the element associated with node \( m \) in the column associated with node \( m \) of the original matrix \([K]\),
- Subtracting each resulting product from the corresponding element in the original vector \( \{b + g\} \), and then
- Substituting zero for all elements in the \( m \)-th column of \([K]\) except the element in the \( m \)-th row (which had already been set to 1).

So, for each node on which a Dirichlet condition is declared,

- The associated element in the vector \( \{b + g\} \) is replaced by the known value at the node corresponding to that element,
- The other elements in the vector \( \{b + g\} \) are adjusted as per Eq. (15.210),
- All elements in the associated row and the associated column of \([K]\) except the element in the intersection of that row and column are set to 0, and
- The element at that intersection is set to 1.

If we have Dirichlet conditions at all bounding nodes, incorporation of these conditions results in replacing the original vector \( \{b + g\} \) with a new vector containing no unknown quantities. The problem is now reduced to solving a fully determined (probably large) set of simultaneous linear equations for the vector \( \{\tilde{\varphi}\} \).

When Neumann conditions are specified, we are given the quantity defined in Eq. (15.72). This gives us enough information to evaluate the integral in Eq. (15.207), not around an entire element but along a portion of the boundary of the region \( \Omega \). We are therefore in a position explicitly to evaluate the integrals symbolized above by \( 3 \rightarrow 0, 0 \rightarrow 1 \), etc. Thus, the \( g \)'s associated with portions of the boundary along which Neumann conditions are specified have values directly determinable from those boundary conditions. If Neumann conditions are specified on all boundaries, all parameters in the equation \([K]\{\tilde{\varphi}\} = \{b + g\}\) except the dependent variable \( \tilde{\varphi} \) are known from the beginning, so again the problem is reduced to solving a fully determined set of simultaneous linear equations.

We can proceed no further in this example without resorting to a specific coding language, so we postpone further discussion to later sections.

### 15.29 Using C to Solve 2D PDEs via an FEM

#### 15.29.1 A General Coding using LAPACK

The process of assembling the system of equations is tedious, if not difficult. The task of solving the resulting large set of simultaneous algebraic equations is to be sure, straightforward, but it can be enormously time-consuming, especially when the region of interest is more complicated than the simple eight-element geometry that we have so far discussed. The task is clearly one for a computer, and it will almost certainly be carried out numerically, not symbolically. Ideally, we would like to input an equation of the form of Eq. (15.70), a suitable definition of the region of interest, and all applicable boundary conditions, assigning to the computer the tasks of (1) constructing the complete equation begun in Eq. (15.205), (2) applying the boundary conditions, and (3) solving the system of equations.

In order to simplify the computer coding for this example and provide a clearer picture of finite element programming without becoming overwhelmed by geometric details, we will impose
several restrictions on the problems that our program can solve. First, we will limit ourselves to a square domain located in the first quadrant with one corner at the origin, such that \(0 \leq x, y \leq L\). Further, to ensure that all elements will be isosceles right triangles (which simplifies computation considerably), we will divide each edge of this square into the same number \(d\) of segments of equal length \(L/d\). In addition, we will restrict \(\alpha_x, \alpha_y, \beta,\) and \(f\) to be constants and, instead of allowing for general boundary conditions, we will assume the constant Dirichlet conditions

\[
\varphi(x, 0) = p_2 ; \quad \varphi(x, L) = p_1
\]  

(15.211)

on the bottom \((y = 0)\) and top \((y = L)\) boundaries, the linear variation

\[
\varphi(0, y) = \frac{p_1 - p_2}{L} y + p_2
\]

(15.212)

between the bottom and top values along the left edge \((x = 0)\), and the Neumann boundary condition

\[
\frac{\partial \varphi}{\partial x}(L, y) = q
\]

(15.213)

with \(q\) constant along the right edge \((x = L)\). Collectively, these assumptions illustrate several different types of boundary condition.

The first segment of the program we wish to write will, then, request input of all the constants except\(^{72}\) \(ND\) (the number of segments on each side), \(M = 2*ND*ND\) (the number of elements), and \(N=(ND+1)*(ND+1)\) the number of nodes) needed in the remainder of the program. Appropriate C statements are\(^{73}\)

```c
#define ND 8 /* Number of segments each side */
#define M ( 2*ND*ND ) /* Determine number of elements */
#define N ( (ND+1)*(ND+1) ) /* Determine number of nodes */
float AL, ALPHAX, ALPHAY, BETA, F, P1, P2, Q;
```

As illustrated in Fig. 15.35, the number of segments \(d\) (\(ND\) in the program) also provides a general way to refer to element numbers and global node numbers for varying numbers of elements.

Next, we calculate the length of each segment on each edge given by the length of the edge divided by the number of segments into which the edge is divided with the statements

```c
float DX;
DX = AL/ND; /* Calculate segment size */
```

\(^{72}\)As explained later, we elect to use the LAPACK routine \texttt{sgesv.f} to solve the linear equation to be constructed. That routine requires as input for several arguments a number of two-dimensional arrays that have the right dimensions for the problem. Since dynamic allocation of two-dimensional arrays in C is an advanced topic, we elect to hard code these parameters in the program so the arrays can be given the proper dimensions for the problem. To be definite, we set \(ND\) equal to 8, and we code all else so that only \(ND\) need be changed to create a program to solve the problem for a different number of segments on each side.

\(^{73}\)Variable declaration statements will be included when the variables are introduced but will be collected at the beginning of the program in a final listing.
15.29. USING C TO SOLVE 2D PDES VIA AN FEM

Figure 15.35: General element and global node numbering based on the number of segments \( d \) into which each edge is divided. Consistent with C conventions, we have here numbered elements and nodes starting at 0.

```
int I, J;
float K[N][N], B[N][1];

for(I=0; I<N; I++) /* Initialize K, B */
{ for(J=0; J<N; J++) K[I][J] = 0.0;
  B[I][0] = 0.0;
}
```

We find the coordinates of all of the nodes, electing to store these coordinates in two vectors, \( X \) and \( Y \), where each entry is the \( x \) or \( y \) coordinate of the node whose global node number corresponds to the index of the entry in the vector. These assignments can be made with two nested for loops, the inner loop incrementing row numbers down a column of nodes and the outer loop incrementing the column number. In this way the nodes are traversed in the sequence in which we numbered them. Further, we can keep track of the correct global node number with a simple counter. To accomplish the assignment of values to \( X \) and \( Y \), we would use the coding

\[74\text{Because } \text{sgesv is written to handle several sets of equations with the same coefficient matrix, each corresponding to a column in } B, \text{ the array } B \text{ must in our context must be defined as a two-dimensional array that happens to have only one column.} \]
Figure 15.36: Simple square domain based on Fig. 15.33 with element numbers (left and right),
global node numbers (left), and local node numbers (right), all starting at 0 as appropriate to the
way arrays are indexed in C.

float X[N], Y[N];
int NCT;
NCT=0;
for(I=0; I<=ND; I++) /* Start row number loop */
{ for(J=0; J<=ND; J++) /* Start column number loop */
  { X[NCT] = I*DX; /* Find x coordinate */
    Y[NCT] = AL - J*DX; /* Find y coordinate */
    NCT=NCT+1; /* Increment counter */
  }
}

We assign global and local node numbers following the pattern in Fig. 15.36. The next step is
to create and store the connectivity matrix, so we can easily convert between local and global node
numbers. As illustrated in Table 15.2, this matrix is a two-dimensional array in which the index
of each row is an element number, and the entries in each row are the global node numbers of the
zeroth, first, and second local nodes of that element. Then, in developing the coding to create this
matrix, we recognize that

- Integer division of the element number by $2*d$ yields the index of the vertical line in Fig. 15.35
  on which the first node of the element lies. Here, the index of the vertical line is simply a count
  of the number of vertical lines to its left, so the first vertical line is at index 0, the second at
  index 1, and the last at index $d$.

- On a given vertical line, the global node number of the local node 0 increases by one every
time the element number increases by two. Therefore, we can in general find the global node
  number of an element’s first local node by
    - performing integer division of the element number by two,
    - adding one (since the sequence of zeroth local nodes begins with global node 1), and
    - adding the index of the vertical line on which the node lies.

- Once the global node number of the zeroth local node of a particular element has been found,
  the remaining two global node numbers for that element follow easily. The way in which they
follow, however, depends on whether the element number is odd or even, as reflected in the coding below.

Thus, the connectivity matrix NCM is created by the statements

```c
int NCM[M][3]; /* For connectivity matrix */
int NVL, IE;
for(IE=0; IE< M; IE++) /* For each element IE: */
{ NVL = IE/(2*ND); /* Find index of vertical line */
  NCM[IE][0] = IE/2 + 1 + NVL; /* Find global number of node 0 */
  if (2*(IE/2) == IE) /* If IE is even */
    NCM[IE][1] = NCM[IE][0] + ND; /* find global number of node 1 */
    NCM[IE][2] = NCM[IE][0] - 1; /* find global number of node 2 */
  else /* If IE is odd */
    NCM[IE][1] = NCM[IE][0] + ND + 1; /* find global number of node 1 */
    NCM[IE][2] = NCM[IE][0]; /* find global number of node 2 */
}
```

The next step is to construct the stiffness matrix \(K^{(c)}\) for a single element. As given in Eq. (15.192), the elements of \([K^{(c)}]\) depend on both the interpolation functions and their derivatives. Since our elements are all isosceles right triangles, the coordinates of all local nodes are related through the leg length, \(\Delta x\). Therefore, we can substitute for \(x_1^{(c)}\) and \(x_2^{(c)}\) in terms of \(x_0^{(c)}\) and \(\Delta x\), and for \(y_1^{(c)}\) and \(y_2^{(c)}\) in terms of \(y_0^{(c)}\) and \(\Delta x\) in the expression for \(K_{ij}^{(c)}\). However, because of the different orientation of odd and even numbered elements, these substitutions are not identical for the two cases. For even numbered elements, we set

\[
x_1^{(c)} = x_0^{(c)} + \Delta x \\
x_2^{(c)} = x_0^{(c)} \\
y_1^{(c)} = y_2^{(c)} = y_0^{(c)} + \Delta x
\]

(15.214)

In contrast, for odd numbered elements, we set

\[
x_1^{(c)} = x_2^{(c)} = x_0^{(c)} + \Delta x \\
y_1^{(c)} = y_0^{(c)} \\
y_2^{(c)} = y_0^{(c)} + \Delta x
\]

(15.215)

With these simplifications, the integrals defining the elements of \([K^{(c)}]\) are much easier to evaluate. Whether the element number \(e\) is even or odd, we elect to do the \(y\) integral first and the \(x\) integral second. In all cases, \(x\) will then run from \(x_0^{(c)}\) to \(x_0^{(c)} + \Delta x\). The limits on \(y\), however, will depend on whether \(e\) is odd or even. In the present mesh, the triangular elements are all right, isosceles triangles and the diagonal lines defining the hypotenuses of these triangles lie at a 45° angle to the horizontal. Thus, when \(e\) is odd (see Fig. 15.35), the limits on \(y\) will run from \(y_0^{(c)}\) to \(y_0^{(c)} + x - x_0^{(c)}\), and we conclude that

\[
\int_{\Omega^{(c)}} \ldots dx dy = \int_{x_0^{(c)}}^{x_0^{(c)} + \Delta x} \left[ \int_{y_0^{(c)}}^{y_0^{(c)} + x - x_0^{(c)}} \ldots dy \right] dx \quad (e \text{ odd})
\]

(15.216)

When \(e\) is even, on the other hand, the limits on \(y\) will instead run from \(y = y_1^{(c)} + x - x_1^{(c)}\) to \(y_1^{(c)} + \Delta x\), and we conclude that

\[
\int_{\Omega^{(c)}} \ldots dx dy = \int_{x_0^{(c)}}^{x_0^{(c)} + \Delta x} \left[ \int_{y_0^{(c)} + \Delta x}^{y_0^{(c)} + x - x_0^{(c)}} \ldots dy \right] dx \quad (e \text{ even})
\]

(15.217)

Because the integrands are no worse than quadratic in \(x\) and \(y\), their evaluation is straightforward, but there are many of them and we elect to invoke a symbol manipulating program.\(^75\) It

\(^75\)Details of this evaluation are described more fully in Appendix 15.F.
turns out that the entries in \([K^{(e)}]\) depend only on the parameters \(\Delta x, \alpha_x,\) and \(\alpha_y,\) and not upon \(e,\) the specific element. In other words, there exist only two distinct elemental stiffness matrices, one for even elements and one for odd elements, namely

\[
K^{(e)}_{\text{odd}} = \begin{bmatrix}
\frac{\beta \Delta x^2 + 6 \alpha_y}{12} & \frac{\beta \Delta x^2 - 12 \alpha_x}{24} & \frac{\beta \Delta x^2}{24} \\
\frac{\beta \Delta x^2 - 12 \alpha_x}{24} & \frac{\beta \Delta x^2 + 6 \alpha_x + 6 \alpha_y}{12} & \frac{\beta \Delta x^2 - 12 \alpha_y}{24} \\
\frac{\beta \Delta x^2}{24} & \frac{\beta \Delta x^2 - 12 \alpha_y}{24} & \frac{\beta \Delta x^2 + 6 \alpha_y}{12}
\end{bmatrix}
\] (15.218)

and

\[
K^{(e)}_{\text{even}} = \begin{bmatrix}
\frac{\beta \Delta x^2 + 6 \alpha_y}{12} & \frac{\beta \Delta x^2}{24} & \frac{\beta \Delta x^2 - 12 \alpha_y}{24} \\
\frac{\beta \Delta x^2}{24} & \frac{\beta \Delta x^2 + 6 \alpha_x}{12} & \frac{\beta \Delta x^2 - 12 \alpha_x}{24} \\
\frac{\beta \Delta x^2 - 12 \alpha_y}{24} & \frac{\beta \Delta x^2 - 12 \alpha_x}{24} & \frac{\beta \Delta x^2 + 6 \alpha_x + 6 \alpha_y}{12}
\end{bmatrix}
\] (15.219)

The coding that constructs these two matrices in our program is

```cpp
float Keven[3,3], Kodd[3,3], BX;
BX = BETA*pow(DX,2); /* Evaluate a common quantity */
Kodd[0][0] = (BX+6*ALPHAX)/12; /* Assign the appropriate value to */
Kodd[1][1] = (BX+6*ALPHAX+6*ALPHAY)/12; /* each K[i,j]. Note that the array */
Kodd[2][2] = (BX+6*ALPHAY)/12; /* are symmetric, and that Kodd */
Kodd[0][1] = (BX-12*ALPHAX)/24; /* includes all of the same values as*/
Kodd[1][0] = Kodd[0][1]; /* Kodd, but in different locations. */
Kodd[1][2] = (BX-12*ALPHAY)/24;
Kodd[2][1] = Kodd[1][2];
Kodd[0][2] = BX/24;
Kodd[2][0] = BX/24;

Keven[0][0] = Kodd[2][2];
Keven[1][1] = Kodd[0][0];
Keven[2][2] = Kodd[1][1];
Keven[0][1] = Kodd[0][2];
Keven[1][0] = Keven[0][1];
Keven[1][2] = Kodd[0][1];
Keven[2][1] = Keven[1][2];
Keven[0][2] = Kodd[1][2];
Keven[2][0] = Keven[0][2];
```

We are now ready to assemble the complete matrix \([K]\), some of whose elements were shown in Eq. (15.205). At the same time, we can also construct the vector \(\{b\}\), whose entries \(b_i^{(e)}\) can be found by using a symbol manipulating program to evaluate the integrals in Eq. (15.193). It turns out that all three elements \(b_1^{(e)}, b_2^{(e)},\) and \(b_3^{(e)}\) have the same value \(f \Delta x^2/6,\) regardless of whether \(e\) is odd or even.\(^{76}\) Following the algorithm described at the end of Section 15.23.4 for assembling the final stiffness matrix, we start by creating a null matrix \([K]\) and a null vector \(\{b\}\) of the proper dimensions. Then, we step through the elements one at a time, at each step adding

\(^{76}\)See again Appendix 15.F.
15.29. USING C TO SOLVE 2D PDES VIA AN FEM

the contributions of the element to the accumulating entries at the proper positions in $[K]$ and $\{b\}$. This end is achieved with the coding

```c
int L1, L2;
for(IE=0; IE<M; IE++) /* Count through element numbers */
{
  L1=2*(IE/2); L2=IE; /* L1=L2 if IE is even */
  for(I=0; I<3; I++)
  {
    /* For each local node of element, place its contributions at the correct locations in K and B */
    B[NCM[IE][I]][0]=B[NCM[IE][I]][0]+F*pow(DX,2)/6;
    for(J=0; J<3; J++)
    {
      if (L1 != L2)
        K[NCM[IE][I]][NCM[IE][J]] = K[NCM[IE][I]][NCM[IE][J]] + Kodd[I][J];
      else
        K[NCM[IE][I]][NCM[IE][J]] = K[NCM[IE][I]][NCM[IE][J]] + Keven[I][J];
    }
  }
}
```

Application of the boundary conditions will complete the setup of the problem. To apply the Dirichlet conditions to the top, bottom, and left edges we must first find an algorithm to identify which nodes are on these edges for various values of $d$. By inspection, we find that, by letting $i$ run from one to $d + 1$, global node numbers on the left boundary are given by $i$, on the top by $(i-1)(d+1)+1$, and on the bottom by $i(d+1)$. Once we know these numbers, we can replace the appropriate equations by the given values of $\hat{\phi}$, and then reflect the influence of this relationship in the other equations as described in Section 15.23.5. The requisite coding is

```c
int NU, NS, NT
float P3
for(I=0; I<ND+1; I++)
{
  NU = I; /* Nodes on the left boundary */
  NS = I*(ND+1); /* Nodes on the top boundary */
  NT = (I+1)*(ND+1) - 1; /* Nodes on the bottom boundary */
  P3 = (P1-P2)/AL * Y[NU] + P2; /* Find values of phi on left boundary */
  for(J=0; J<N; J++)
  {
    K[NS][J] = 0.0; /* Set rows in K to zero where value */
    K[NT][J] = 0.0; /* of phi is known */
    K[NU][J] = 0.0;
    B[NS][0] = P1; /* Set values of b */
    B[NT][0] = P2;
    B[NU][0] = P3;
    /* Reflect influence of known values of phi in the other equations */
    if(J != NS) B[J][0] = B[J][0] - K[J][NS]*P1;
    K[J][NS] = 0.0;
    if(J != NT) B[J][0] = B[J][0] - K[J][NT]*P2;
    K[J][NT] = 0.0;
    if(J != NU) B[J][0] = B[J][0] - K[J][NU]*P3;
    K[J][NU] = 0.0;
  }
  K[NS][NS] = 1.0; /* Set the appropriate entry to 1 in */
  K[NT][NT] = 1.0; /* the rows where phi is known */
  K[NU][NU] = 1.0;
}
```
Since the Neumann condition is specified only on the right edge, the unit normal vector \( \hat{n} \) in Eqs. (15.72) and (15.194) becomes simply \( \hat{i} \). Also, as shown in Section 15.23.5, we need worry only about segments on the boundary of \( \Omega \) when integrating to find \( g_i^{(e)} \). These two items and the fact that all elements with boundaries on the right edge have odd numbers reduce Eq. (15.194) to

\[
g_i^{(e)} = \int_{g_i^{(e)}}^{g_i^{(e)} + \Delta x} N_1^{(e)} \alpha_x \frac{\partial \tilde{\varphi}}{\partial x} \, dy \tag{15.220}
\]

Since we know that \( \alpha_x (\partial \tilde{\varphi} / \partial x) = q \) and that \( x = x_1^{(e)} + \Delta x \), we can easily evaluate this expression, provided we assume \( \alpha_x \) can be treated as a constant over the area of each element. Again invoking a symbol manipulating program,\(^{77}\) we find that

\[
g_0^{(e)} = 0 \quad ; \quad g_1^{(e)} = \frac{q \Delta x}{2} \quad ; \quad g_2^{(e)} = \frac{q \Delta x}{2} \tag{15.221}
\]

The contribution of each node to the complete \( \{g\} \) vector will be twice this value, since it consists of two parts, one from each right boundary element that contains the node. We don’t need \( g_0^{(e)} \) since no node on the right edge is a first local node. The necessary additional coding, including the consolidation of the inhomogeneities into a single vector, is

```c
float G[N][0];
int NND;
for(I=0; I<ND; I++) /* Nodes on right boundary */
{ NND = ND*(ND+1)+1;
  G[NND][0] = Q*DX;
}
for(I=0; I<ND; I++) B[I][0] = B[I][0] + G[I][0]; /* {b} ==> {b+g} */
```

At this point, \( K \) contains the coefficient matrix and \( B \) the inhomogeneities for the system of simultaneous linear equations we wish to solve.

For the final solution of this system of equations, we elect to use the LAPACK routine `sgesv.f`, which is a single-precision (s) routine for solving general (ge) simultaneous linear algebraic equations.\(^{79}\) Properly set up, FORTRAN routines can be called from C programs. The appropriate C statements are

```c
int INFO, IPIV[N];
I=1; J=N;
sgesv_(&J, &I, &K, &J, &IPIV, &B, &J, &INFO);
```

Here,

- The first argument, \( N \), on input is the number of equations; it is not changed on output.
- The second argument, \( 1 \), on input is the number of columns in \( B \); it is not changed on output.
- The third argument, \( K \), on input is the coefficient matrix; on output it contains the LU decomposition of the input matrix.
- The fourth argument, \( N \), on input is the number of rows in \( AK \); it is not changed on output.
- The fifth argument, \( IPIV \), on input is not used; on output it is an integer array conveying the exchanges of rows/columns made in the process of working out the solution. It must be dimensioned as a one-dimensional integer array of size equal to the number of equations.

---

\(^{77}\)See Appendix 15.F once more.

\(^{78}\)Systems whose coefficient matrix is not necessarily tridiagonal or symmetric.

• The sixth argument B on input is the array of inhomogeneities; on output it is the array of solutions.
• The seventh argument, N, on input is the number of rows in B; it is not changed on output.
• The eighth argument, INFO, on input is not used; on output its value conveys whether or not the solution is successful.

More detailed information about these variables is compiled in the comments at the beginning of the LAPACK file sgesv.f.

At this point, B contains the (approximate) solution to the boundary value problem defined in Section 15.1.8, though it is in a vector of values, the index of each being the node number. Consequently, before writing the solution to a file, we must rearrange the values in B into a two-dimensional array whose elements are arranged as the nodes in the geometry of the problem. The coding

```c
float PHI[ND+1][ND+1]; /* For solution */
NCT=0; /* Initialize a counter variable */
for(I=0; I<=ND; I++) /* Use nested loops to write all */
{ for(J=0; J<=ND; J++) /* entries into PHI */
  { PHI[I][J] = B[NCT][0];
    NCT = NCT + 1;
  }
}
```

accomplishes that rearrangement. Since the nodes are, in this example, uniformly spaced over the region of the problem, we need not be concerned about knowing the nodal coordinates for purposes of plotting the solution. The coordinates of the node corresponding to each value in the two-dimensional array PHI are proportional to the indices of that value in the array.

Finally, we need to write this solution out to a file so we can transfer it easily to a visualization program for graphical display. Essentially, we need to open a file, write the solution and its dimensions to the file, and close the file. Statements achieving those objectives are

```c
FILE *fptr; /* For file pointer */
fptr = fopen( "fem2dla_c.dat", "w" );
fprintf( fptr, "%d
", ND+1 );
for(J=0; J<=ND; J++)
{ for(I=0; I<=ND; I++)
  { fprintf(fptr, "%10.4f", PHI[I][J] );
  }
fprintf( fptr, "\n" );
}
fclose( fptr );
```

Our program is now complete. All of the preceding C code has been incorporated into the program fem2dla.c, a commented listing of which can be found in Appendix 15.G.6. (The file itself can be copied from the directory $HEAD/cc.)

We are at last ready to compile and link this program with the needed LAPACK routines to create the executable file we can use to solve specific problems. The way to accomplish that objective depends on how things are set up at the local site. In particular, the file defining sgesv.f and files defining all routines supporting sgesv.f must be available in some way. At Lawrence,81

80 The sequence in which the values are written to the file has been chosen so that, when printed, the value in the file will be arranged on the page with the left, right, top, and bottom edges corresponding to the arrangement we have assumed in the set-up of the problem.

81 Your Local Guide will describe the procedures at your site.
those files are all compiled and collected into the shared library named sgesvlib.so. Thus, copying that file into the directory containing fem1dla.f and, working in a command window with that directory as its default, we execute the statement

```
cc -o fem2dla.xf fem2dla.c sgesvlib.so
```
to create the executable file fem2dla.xf.

### 15.29.2 An Example: Isotropic Heat Flow

#### 15.29.2.1 Setting the Problem

We will now demonstrate fem2d.f by applying it to an equation in isotropic heat flow. The steady-state temperature \( u(x, y) \) in a square plate of uniform composition is a solution to the two-dimensional Laplace equation

\[
\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \tag{15.222}
\]
to which Eq. (15.70) reduces when we replace \( \tilde{\varphi} \) with \( u \) and set

\[
\alpha_x = 1 ; \quad \alpha_y = 1 ; \quad \beta = 0 ; \quad f = 0 \tag{15.223}
\]

We will take the plate to be ten units square \((L = 10)\), and position it in the first quadrant with a corner at the origin, and we will apply the boundary conditions

\[
u(x, 0) = 0 ; \quad u(x, 10) = 100 ; \quad u(0, y) = 10y ; \quad \frac{\partial u}{\partial x}(10, y) = 0 \tag{15.224}
\]
i.e., \( p_2 = 0, p_1 = 100, L = 10, \) and \( q = 0 \).

#### 15.29.2.2 Running the C Program

With these choices, we run the program and solve the problem with the single statement

```
./fem2dla.xf
```

At the several prompts generated by this statement, we enter the values

```
Enter length of side (L): 10
Enter number of segments (d): 8
Enter alpha_x: 1
Enter alpha_y: 1
Enter beta: 0
Enter f: 0
Enter value for top edge: 100
Enter value for bottom edge: 0
Enter q: 0
```

Each line appears one at a time, and waits for us to input the requested parameter. After the last parameter has been entered, the program generates a solution and writes it to the file fem2dla.f.dat in the default directory. For this input (and with a bit of editing in the values to remove unnecessary zeros so as to squeeze what are two lines in the file into one line in the display), the resulting file is
15.29. USING C TO SOLVE 2D PDES VIA AN FEM

which—intuitively (and somewhat accidentally because of the simplicity of the problem)—is exactly correct.

15.29.2.3 Displaying the Solution Graphically with IDL

To display the contents of this file graphically using IDL, we would import the file and produce a surface plot with the statements

```
IDL> openr, 1, 'fem2dla_c.dat'
IDL> readf, 1, n
IDL> soln = fltarr(n,n)
IDL> readf, 1, soln
IDL> close, 1
IDL> x = 10.0*findgen(n)/(n-1.0)
IDL> surface, reverse(soln, 2), x, x, charsize=2.0, thick=5.0, $ xtitle='x', ytitle='y'
```

The resulting graph is shown in Fig. 15.37. This figure shows that the temperature varies linearly throughout the plate, a result that is entirely consistent with our intuition: the steady-state temperature in the plate should vary linearly between the extremes on opposite edges.

15.29.2.4 Displaying the Solution Graphically with MATLAB

To display the contents of this file graphically using MATLAB, we would import the file and produce a surface plot with the statements

```
id = fopen('fem2dla_c.dat', 'r' );
n = fscanf( id, '%d', 1 );
soln = fscanf( id, '%f', [n,n] );
status = fclose( id );
x= 10.0*[ 0 : 8 ]/8.0;
mesh( x, x, rot90(soln), 'EdgeColor', 'black', 'LineWidth', 3 );
set(gca,'fontsize',16)
xlabel('x','fontsize',20); ylabel('y','fontsize',20)
zlabel('u','fontsize',20)
view([45,30])
```

The resulting graph is shown in Fig. 15.38. This figure shows that the temperature varies linearly throughout the plate, a result that is entirely consistent with our intuition: the steady-state temperature in the plate should vary linearly between the extremes on opposite edges.
CHAPTER 15. SOLVING PARTIAL DIFFERENTIAL EQUATIONS

Figure 15.37: The steady-state temperature distribution of the isotropic plate. This graph was produced with IDL from a data file generated by a C program.

Figure 15.38: The steady-state temperature distribution of the isotropic plate. This graph was produced with MATLAB from a data file generated by a C program.
15.29. USING C TO SOLVE 2D PDES VIA AN FEM

Figure 15.39: The steady-state temperature distribution of the isotropic plate. This graph was produced with OCTAVE from a data file generated by a C program.

15.29.2.5 Displaying the Solution Graphically with OCTAVE

To display the contents of this file graphically using OCTAVE, we would import the file and produce a surface plot with the statements

```octave
id = fopen('fem2dla_c.dat', 'r' );
n = fscanf( id, '%d', 1 );
soln = fscanf( id, '%f', [n,n] );
status = fclose( id );
x= 10.0*[ 0 : 8 ]/8.0;
mesh( x, x, rot90(soln), 'EdgeColor', 'black', 'LineWidth', 3 );
set(gca,'fontsize',16)
xlabel('x','fontsize',20); ylabel('y','fontsize',20)
zlabel('u','fontsize',20)
view(azim=45,elev=30)
```

The resulting graph is shown in Fig. 15.39. This figure shows that the temperature varies linearly throughout the plate, a result that is entirely consistent with our intuition: the steady-state temperature in the plate should vary linearly between the extremes on opposite edges.

15.29.2.6 Displaying the Solution Graphically with PYTHON

To display the contents of this file graphically using PYTHON, we would import the file with the statements

```python
f = open( 'fem2dla_c.dat', 'r' )
n = int( f.readline() )
u=[]
```
Figure 15.40: The steady-state temperature distribution of the isotropic plate. This graph was produced with PYTHON from a data file generated by a C program.

```python
for j in range(n):
    row = [float(tmp) for tmp in f.readline().split()]
    u.append(row)
f.close()

import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

xx = np.linspace(0.0,10.0,n)
x,y = np.meshgrid(xx,xx)
fig1 = plt.figure(1)
ax1 = plt.axes(projection='3d')
u1 = np.flip(u, axis=0)
ax1.plot_surface(x, y, u1, color='white', shade=False, edgecolor='black')
ax1.set_xlabel('x'); ax1.set_ylabel('y'); ax1.set_zlabel('u')
plt.show()
```

The resulting graph is shown in Fig. 15.40. This figure shows that the temperature varies linearly throughout the plate, a result that is entirely consistent with our intuition: the steady-state temperature in the plate should vary linearly between the extremes on opposite edges.

### 15.29.3 Assessing Accuracy

When working with numerical approximation techniques such as finite element analysis, we must always be wary of the accuracy of results. When investigating a problem with a less evident solution,
one can test the results by performing additional analysis using a larger number of elements. We could, for example, generate a solution with a larger number of elements by editing the program to change the value of \( ND \) to, say, 17, recompiling the program, and then running it with the (same) input, specifically

```
./fem2dla.xc
Enter length of side (L): 10
Enter alpha_x: 1
Enter alpha_y: 1
Enter beta: 0
Enter f: 0
Enter value for top edge: 100
Enter value for bottom edge: 0
Enter q: 0
```

which will produce a solution on a 17 \( \times \) 17 grid for comparison with the solution produced on the 9 \( \times \) 9 grid. We have, in fact, already made that comparison in a previous section (or previous sections), and we will not here repeat that discussion.

### 15.30 Exercises

Note: In these exercises, we refer to the programs developed in the text without appending the file type, adopting this approach to leave to you the choice of which language to use in addressing the exercise.

15.1. Recast \texttt{fdm1d} to solve the ODE of Eq. (pde:diffeq) when \( f(x) = kx \), the boundary conditions are

\[
\frac{\partial \varphi}{\partial x}(0) = p ; \quad \varphi(L) = q
\]

and \( \alpha, \beta, k, p, q, \) and \( L \) are constants whose values are to be read in at execution time. Then, using your command file, explore the solution to the equation for various values of \( k \) when

\[
\alpha = -4.0 ; \quad \beta = 4.0 ; \quad L = 10.0 ; \quad p = q = 0
\]

Optional: (a) Find an analytic solution to the equation in this exercise and compare the exact results with the approximate results generated by your modification of \texttt{fdm1d}. (b) Find the points at which the solution has the value zero, both starting with the solution obtained in the main exercise and working from the exact solution obtained in optional part (a).

15.2. Suppose the one-dimensional string of length \( l \) discussed in Section 15.1.1 hangs vertically and is acted on by gravity. Suppose that \( u(x, t) \) and \( v(x, t) \) give the transverse (horizontal) and longitudinal (vertical) displacements of the particle of the string nominally located at \( x \), which is measured downward from the top of the string. (a) Examine the forces acting on this string, deduce the general equations for both longitudinal and transverse motion of the string, and show ultimately that, if the motion is entirely transverse and the amplitude of the motion is small, the equations reduce to

\[
\rho(x) \frac{\partial^2 u}{\partial t^2} = \frac{\partial}{\partial x} \left( \tau(x) \frac{\partial u}{\partial x} \right) ; \quad \tau(x) = -g \int_x^l \rho(x') \, dx'
\]

where \( \rho(x) \) is the mass per unit length of the string. (b) Taking \( \rho \) to be constant, show that the tension \( \tau(x) \), which is simply equal to the weight of the string below \( x \), is given by \( \tau(x) = \rho g(l-x) \). With this restriction, the equation of motion then becomes

\[
\frac{\partial^2 u}{\partial t^2} = \frac{\partial}{\partial x} \left( (l-x) g \frac{\partial u}{\partial x} \right)
\]

(c) Recast this equation in dimensionless form by introducing the variables \( \overline{x} = x/l \) and \( \overline{t} = t \sqrt{g/l} \). (d) Suppose you seek a sinusoidal solution for which \( u(\overline{x}, \overline{t}) = f(\overline{x}) \cos \omega \overline{t} \). Find the ODE satisfied
CHAPTER 15. SOLVING PARTIAL DIFFERENTIAL EQUATIONS

Suppose you know three points \((x_0, f_0)\), \((x_1 = x_0 + \Delta x, f_1)\), and \((x_2 = x_0 + 2 \Delta x, f_2)\) on a curve and you wish to estimate the derivative of the corresponding function at \(x = x_0\) by a method that is more accurate than simply the forward difference formula \((f_1 - f_0)/\Delta x\). You might fit the parabola \(f(x) = Ax^2 + Bx + C\) through the three points and then approximate the derivative of the actual function as the derivative of this parabola at the point \(x = x_0\). Show that this approach yields the formula

\[
\frac{df}{dx} \bigg|_{x=x_0} = \frac{-3f_0 + 4f_1 - f_2}{2 \Delta x}
\]

which is a higher-order forward difference approximation than the one used in the text. Hint: Set up the three equations \(f_s = Ax_r^2 + Bx_r + C\), solve those equations for \(A, B, C\), and note that \(df/dx = 2Ax + B\). You may find a symbolic manipulating program of substantial assistance.
15.7. Consider the differential equation and boundary conditions
\[
\frac{d^2u}{dx^2} + k^2u = 0 \quad ; \quad u(0) = u(L) = 0
\]

(a) Find the analytic solution to this problem and identify the special values of \(k\) that permit non-trivial solutions. (b) Show that the finite difference approach to the problem leads to the matrix eigenvalue problem \(Au = -k^2L^2u/N^2\) where \(N\) is the number of equal-length segments into which the length \(L\) of the domain is divided and

\[
A = \begin{pmatrix}
-2 & 1 & 0 & \cdots & 0 & 0 & 0 \\
1 & -2 & 1 & \cdots & 0 & 0 & 0 \\
0 & 1 & -2 & \cdots & 0 & 0 & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & \cdots & -2 & 1 & 0 \\
0 & 0 & 0 & \cdots & 1 & -2 & 1 \\
0 & 0 & 0 & \cdots & 0 & 1 & -2
\end{pmatrix} \quad ; \quad u = \begin{pmatrix}
u_1 \\
u_2 \\
u_3 \\
\vdots \\
u_{N-3} \\
u_{N-2} \\
u_{N-1}
\end{pmatrix}
\]

Note that, when the domain \(0 \leq x \leq L\) is divided into \(N\) segments, there will be \(N + 1\) nodes ranging from \(x_0 = 0\) to \(x_{N+1} = L\). Because, along the way to a solution, the boundary conditions result in the rows and columns associated with these two nodes being deleted, these matrices will have only \(N - 1\) rows and columns. (c) Taking \(N = 100\), use an available numeric processing program like IDL, MATLAB, OCTAVE, or PYTHON to find the first several eigenvalues \(k_n\) and compare your results with the values found in part (a).

15.8. Consider the differential equation and boundary conditions
\[
\frac{d^2u}{dx^2} + k^2u = 0 \quad ; \quad u(0) = u(L) = 0
\]

(a) Find the analytic solution to this problem and identify the special values of \(k\) that permit non-trivial solutions. (b) Show that the finite element approach to the problem leads to the generalized eigenvalue problem \(Au = -k^2L^2Bu/6N^2\) where \(N\) is the number of equal-length elements into which the length \(L\) of the domain is divided and

\[
A = \begin{pmatrix}
-2 & 1 & 0 & \cdots & 0 & 0 & 0 \\
1 & -2 & 1 & \cdots & 0 & 0 & 0 \\
0 & 1 & -2 & \cdots & 0 & 0 & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & \cdots & -2 & 1 & 0 \\
0 & 0 & 0 & \cdots & 1 & -2 & 1 \\
0 & 0 & 0 & \cdots & 0 & 1 & -2
\end{pmatrix} \quad ; \quad B = \begin{pmatrix}
4 & 1 & 0 & \cdots & 0 & 0 & 0 \\
1 & 4 & 1 & \cdots & 0 & 0 & 0 \\
0 & 1 & 4 & \cdots & 0 & 0 & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & \cdots & 4 & 1 & 0 \\
0 & 0 & 0 & \cdots & 1 & 4 & 1 \\
0 & 0 & 0 & \cdots & 0 & 1 & 4
\end{pmatrix}
\]

and

\[
u = \begin{pmatrix}
u_1 \\
u_2 \\
u_3 \\
\vdots \\
u_{N-3} \\
u_{N-2} \\
u_{N-1}
\end{pmatrix}
\]

Note that, when the domain \(0 \leq x \leq L\) is divided into \(N\) segments, there will be \(N + 1\) nodes ranging from \(x_0 = 0\) to \(x_{N+1} = L\). Because, along the way to a solution, the boundary conditions result in the rows and columns associated with these two nodes being deleted, these matrices will have only \(N - 1\) rows and columns. (c) Taking \(N = 100\), use an available numeric processing program like IDL, MATLAB, OCTAVE, or PYTHON to find the first several eigenvalues \(k_n\) and compare your results with the values found in part (a).
15.9. When $\alpha$, $\beta$, and $f$ are constants, Eqs. (15.67), (15.68), and (15.69) can be solved analytically. Show, for example, that the solution to this boundary value problem is given by

$$\varphi(x) = A \sin \lambda x + \left( p - \frac{f}{\beta} \right) \cos \lambda x + \frac{f}{\beta}$$

where

$$A = \frac{(q - \gamma f/\beta) + (p - f/\beta)(\alpha \lambda \sin \lambda L - \gamma \cos \lambda L)}{\alpha \lambda \cos \lambda L + \gamma \sin \lambda L}$$

and $\lambda = \sqrt{-\beta/\alpha}$ when $\beta/\alpha < 0$. Using graphical displays in particular, compare the analytic solution in this exercise with the solution obtained numerically by finite difference and finite element approaches. *Optional:* Find corresponding solutions when $\beta/\alpha = 0$ and $\beta/\alpha > 0$.

15.10. Suppose a linear element $e$ is characterized by three nodes at $x_1^e$, $x_2^e$, and $x_3^e$. Further, let the solution $\varphi^e(x)$ on that element be approximated by the quadratic function

$$\varphi^e(x) = a^e + b^e x + c^e x^2$$

(a) Find the constants $a^e$, $b^e$, and $c^e$ that will make this function match the specific values $\varphi_1^e$, $\varphi_2^e$, and $\varphi_3^e$ at the points $x = x_1^e$, $x_2^e$, and $x_3^e$, respectively. (b) Substituting these values into the above expression and grouping terms appropriately, cast the result in the form

$$\varphi^e(x) = \sum_{i=1}^{3} c_i^e N_i^e(x)$$

and show that the shape functions $N_i^e(x)$ appropriate to this three-noded linear element are

$$N_1^e(x) = \frac{1}{\Delta} \begin{vmatrix} 1 & x & x^2 \\ x_1^e & (x_1^e)^2 & (x_1^e)^3 \\ x_2^e & (x_2^e)^2 & (x_2^e)^3 \end{vmatrix} = \frac{(x_2^e - x)(x_3^e - x)}{(x_2^e - x_1^e)(x_3^e - x_1^e)}$$

$$N_2^e(x) = \frac{1}{\Delta} \begin{vmatrix} 1 & x & x^2 \\ x_1^e & (x_1^e)^2 & (x_1^e)^3 \\ x_3^e & (x_3^e)^2 & (x_3^e)^3 \end{vmatrix} = \frac{(x - x_1^e)(x_3^e - x)}{(x_2^e - x_1^e)(x_3^e - x_2^e)}$$

$$N_3^e(x) = \frac{1}{\Delta} \begin{vmatrix} 1 & x & x^2 \\ x_1^e & (x_1^e)^2 & (x_1^e)^3 \\ x_2^e & (x_2^e)^2 & (x_2^e)^3 \end{vmatrix} = \frac{(x - x_2^e)(x - x_1^e)}{(x_3^e - x_1^e)(x_3^e - x_2^e)}$$

where

$$\Delta = \begin{vmatrix} 1 & x_1^e & (x_1^e)^2 \\ 1 & x_2^e & (x_2^e)^2 \\ 1 & x_3^e & (x_3^e)^2 \end{vmatrix}$$

Finally, (c) show that—with $\xi = (x - x_1^e)/l^e$—these functions reduce to

$$N_1^e(x) = 2(\xi - 1) \left( \xi - \frac{1}{2} \right) ; \quad N_2^e(x) = 4\xi(1 - \xi) ; \quad N_3^e(x) = 2\xi \left( \xi - \frac{1}{2} \right)$$

when $x_2^e$ is midway between $x_1^e$ and $x_3^e$, i.e., when $x_2^e = x_1^e + \frac{1}{2} l^e$ and $x_3^e = x_1^e + l^e$, and (d) obtain graphs of these three shape functions over the interval $0 < \xi < 1$. *Hint:* You may find a symbolic manipulating program to be useful at many points in this problem.

15.11. Accepting the shape functions given in part (c) of Exercise 15.10, (a) find the $3 \times 3$ matrices

$$\int_{x_1^e}^{x_2^e} \frac{dN_i}{dx} \frac{dN_j}{dx} dx \quad \text{and} \quad \int_{x_1^e}^{x_3^e} N_i N_j dx$$

and then construct the $3 \times 3$ matrix whose elements are

$$K_{ij}^{e} = \alpha^{e} K_{ij}^{1} + \beta^{e} K_{ij}^{2}$$
15.30. EXERCISES

287

15.12. Continuing with the circumstances of the previous two problems, suppose the region of interest is divided into three three-noded elements, \( e = 1, 2, 3 \) with global nodes 1, 2, 3, 4, 5, 6, 7, nodes 1, 2, and 3 in element 1, nodes 3, 4, and 5 in element 2, and nodes 5, 6, and 7 in element 3. Suppose node 2 is midway between nodes 1 and 3, node 4 is midway between nodes 3 and 5, and node 6 is midway between nodes 5 and 7, but do not suppose the lengths of the three elements are the same. Following the pattern in Section 15.9.4, assemble the elemental equations for these three elements into an equation for the whole system analogous to Eq. (15.130) if the solution is required to satisfy the boundary conditions of Eqs. (15.68) and (15.69).

15.13. Recast \texttt{femid} so it will generate a solution to Eq. (15.67) when

- \( \alpha \) is a (positive) constant and \( \beta = 0 \),
- \( f(x) \) varies with position in accordance with \( f(x) = Ae^{-\sigma(x-L/2)^2} \), with \( A \) and \( \sigma \) constants, and
- the solution is to have the fixed value \( \varphi = 0 \) at \( x = 0 \) and the fixed value \( \varphi = 100 \) at \( x = L \), i.e., we have a source that is concentrated near the middle of the region of interest and we impose Dirichlet boundary conditions on both ends.

Then, explore the character of the solutions for various values of \( \sigma \). Note: This solution to this problem corresponds physically to the steady state temperature in a one-dimensional rod whose ends are maintained at the fixed temperatures 0 and 100, respectively, and whose middle is heated with a source that provides a constant energy input.

15.14. Repeat the previous problem but pursue its solution this time by using elements of varying length designed to recognize that, especially if the Gaussian function is sharply peaked in the center, it might be wise to use smaller elements in that region.

15.15. (a) Solve Eq. (15.179) for the constants \( a^{(e)} \), \( b^{(e)} \), and \( c^{(e)} \) and then verify the expressions given in Eqs. (15.182)–(15.184) for the shape functions. Remember that symbol manipulating programs are available. (b) For each node \( i \), verify that the function \( N^{(e)}(x, y) \) as given by Eqs. (15.182) and (15.184) is zero not only at the two nodes not identified by its index but also along the entire line to produce the (elemental) stiffness matrix analogous to Eq. (15.98). (b) Find the three-element vector whose elements are

\[ b_i^{(e)} = f^{(e)} \int_{x_i^{(e)}}^{x_i^{(e)}} N_i^{(e)} \, dx \]

analogous to Eq. (15.99) and the three element vector whose elements are

\[ g_i^{(e)} = \alpha^{(e)} N_i^{(e)} \frac{d \tilde{\varphi}}{dx} \bigg|_{x=x_i^{(e)}} - \alpha^{(e)} N_i^{(e)} \frac{d \tilde{\varphi}}{dx} \bigg|_{x=x_i^{(e)}}. \]

analogous to Eq. (15.100). By the time you are done, you should discover that the elemental solution is to have the fixed value \( \varphi = 0 \) at \( x = 0 \) and the fixed value \( \varphi = 100 \) at \( x = L \), i.e., we have a source that is concentrated near the middle of the region of interest and we impose Dirichlet boundary conditions on both ends.
joining those two nodes. **Hint:** Set the determinant in the numerator of the expression giving \( N^{(c)} \) to zero, thereby obtaining the equation of a line in the plane. Verify that that line is, in fact, the line joining the two described nodes. (c) Find the functions to which these functions reduce when \((x_0, y_0) = (-1, 0), \ (x_1, y_1) = (1, 0), \ (x_2, y_2) = (0, 1)\) and demonstrate that each is zero where it is supposed to be zero and one where it is supposed to be one.

### 15.16

Consider a four-node element in two dimensions. Let the nodes be at \((x_0, y_0), \ (x_1, y_1), \ (x_2, y_2), \) and \((x_3, y_3)\), and take the interpolating function to be \( \tilde{\varphi} = a + bx + cy + dx y \). (a) Find the four shape functions in general terms, showing that those functions can be expressed in the form

\[
N_0(x, y) = \frac{1}{\Delta} \begin{vmatrix}
1 & x & y & xy \\
1 & x_1 & y_1 & x_1 y_1 \\
1 & x_2 & y_2 & x_2 y_2 \\
1 & x_3 & y_3 & x_3 y_3
\end{vmatrix}
\]

and in similar forms for \( N_1(x, y) \), \( N_2(x, y) \), and \( N_3(x, y) \). (b) Then, show that, if the nodes lie at the corners of a square of side \( s \), i.e., the nodes are—in order—at \((0, 0), \ (s, 0), \ (s, s), \) and \((0, s)\), the shape functions expressed in terms of the variables \( \xi = x/s \) and \( \eta = y/s \) are

\[
N_0(\xi, \eta) = (1 - \xi)(1 - \eta) ; \quad N_1(\xi, \eta) = \xi (1 - \eta) ; \quad N_2(\xi, \eta) = \xi \eta ; \quad N_3(\xi, \eta) = \eta (1 - \xi)
\]

(c) Finally generate surface graphs of the four functions and verify that the shape function associated with each node is zero everywhere along the two edges that intersect at the diagonally opposite node. **Hints:** (1) Remember that symbol manipulating programs are available. (2) Some results along the way to a solution are very involved. Presentation of the intermediate results can certainly be suppressed.

### 15.17

Find the solution for steady state temperature when a square as in the text has its lower edge maintained at 0, the temperature on its left edge rises linearly from 0 to 50, that on its upper edge rises linearly from 50 to 100, and its right edge is insulated.

### 15.18

Proof that the full discretization of the wave equation as described towards the end of Section 15.16.1 leads to an unstable method unless \( \alpha \leq 1 \) is extremely difficult. (A proof is worked out in pages 16–29 of *Finite Difference Methods for Partial Differential Equations*, George Forsythe and Wolfgang Wasow (John Wiley and Sons, New York, 1960).) This exercise asks not for that proof but only that you obtain evidence supporting the existence of that instability by recasting `fdmwaved1` to solve the wave equation subject to the boundary conditions \( u(0, t) = u(l, t) = 0 \) and the initial conditions \( u(x, 0) = 0, \ \partial u(x, 0)/\partial x = 0 \). The solution should, of course, be zero at all subsequent times, since we have started the string in its equilibrium position with zero velocity. Now, suppose that a computer roundoff error occurs such that, instead of being zero at all nodes, \( u(x, \Delta t) \) is everywhere *except at one node near the middle of the string* and, at that node \( u(x, \Delta t) \) mistakenly acquires the value 1 (one). Use your program to solve this problem for choices of the parameters that make \( \alpha = 0.5 \) and solve it again for other choices that make \( \alpha = 1.5 \). Look at the solution with print or plot frequencies 1 and compare the way in which that one disruption of a value at the end of the first time step propagates forward in time for the two values of \( \alpha \).

### 15.19

Proof that the full discretization of the diffusion equation as described towards the end of Section 15.16.2 leads to an unstable method unless \( \gamma \leq 1/2 \) is extremely difficult. (A proof is worked out in pages 92–98 of *Finite Difference Methods for Partial Differential Equations*, George Forsythe and Wolfgang Wasow (John Wiley and Sons, New York, 1960).) This exercise asks not for that proof but only that you obtain evidence supporting the existence of that instability by recasting `fdmdiffus1d` to solve the diffusion equation subject to the boundary conditions \( u(0, t) = u(l, t) = 0 \) and the initial condition \( u(x, 0) = 0 \). The solution should, of course, be zero at all subsequent times, since we have started the temperature distribution with its equilibrium values. Now, suppose that a computer roundoff error occurs such that, instead of being zero at all nodes, \( u(x, \Delta t) \) is everywhere *except at one node near the middle of the rod* and, at that node \( u(x, \Delta t) \) mistakenly acquires the value 1 (one). Use your program to solve this problem for choices of the parameters that make \( \gamma = 0.25 \) and other choices that make \( \gamma = 1.0 \). Look at the solution with print or plot frequency 1 and compare the way in which that one disruption of a value at the end of the first time step propagates forward in time for the two values of \( \gamma \).
15.20. The inhomogeneous Helmholtz equation in two-dimensional Cartesian coordinates is

\[ \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + k^2 u = r(x, y) \]

where \( k^2 \) is a constant and \( r(x, y) \) is the inhomogeneity. Apply finite difference methods to show that

\[ u_{i,j} \approx \frac{u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - \Delta x^2 r_{i,j}}{4 - k^2 \Delta x^2} \]

Here, \( u_{i,j} = u(x_i, y_j) \), the spacing between consecutive values of \( x \) is \( \Delta x \), the spacing between consecutive values of \( y \) is \( \Delta y \) and \( \Delta y = \Delta x \). This result could be used in an iterative approach to solving the inhomogeneous Helmholtz equation. \textit{Note}: If \( k^2 = 0 \), the equation of this exercise reduces to the inhomogeneous Laplace equation, i.e., to the Poisson equation. If, on the other hand, \( r(x, y) = 0 \), then this equation reduces to the (homogeneous) Helmholtz equation.

15.21. Starting with \( v = \sqrt{RT/m_\alpha} \), set \( T = T_0 + \Delta T \) and expand about \( T_0 \) and show that for small variations \( \Delta T \) about this base value, the speed of sound varies linearly with \( \Delta T \).

15.22. In the text, we used the formula in Eq. (15.168) as the basis for an iterative algorithm for solving Laplace’s equation, i.e., for relaxing the initial guess to a final solution. This formula estimates the next iterate at a particular node as the average of that node’s four nearest neighbors. An alternative approach, known as over-relaxation, includes a contribution from the current value in the node itself, specifically

\[ u_{i,j} = (1 - \alpha) u_{i,j} + \alpha \frac{u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1}}{4} \]

where \( \alpha = 1 \) reduces this equation to Eq. (15.168) and \( 1 \leq \alpha < 2 \). While the optimal choice of \( \alpha \) depends on the size of the grid used, in general using a value of \( \alpha > 1 \) will speed the convergence of the iterative process to an acceptably accurate solution. Recast \textit{fdmlap2d} to (1) accept a value of \( \alpha \) as input, (2) implement over-relaxation, (3) monitor the absolute value of the node-by-node change between successive iterates, and (4) print out the maximum value of that change at the end of each iteration. Then, use your program to explore the impact of various values of \( \alpha \) on the rate of convergence for the specific example treated in the text.

15.23. Recast \textit{fdmlap2d} so that iteration stops when the largest change occurring at any single node from one iterate to the next does not exceed an externally prescribed tolerance. To avoid all possibility of an infinite loop, you should halt iteration \textit{either} when the tolerance has been reached (or exceeded) \textit{or} when some prescribed number of iterations has taken place. Code so that your program displays the actual tolerance achieved. Further, if execution terminates because the prescribed tolerance is \textit{not} achieved, your program should print a message that alerts you to the fact that the prescribed tolerance was not achieved. Test your program with the same example as was used in the text. \textit{Hints}: Before starting an iteration, set a variable, say \texttt{maxch}, equal to zero. As you calculate a new value for each node in the iteration, store the result in a temporary variable so you can compare that value with the old value it will replace, updating \texttt{maxch} to the absolute value of the difference between the new and the old values \textit{if and only if} that difference exceeds the difference already stored in \texttt{maxch}. Then, substitute the new value for the old in the array containing the evolving solution and go on to the next node. Once the iteration is completed, \texttt{maxch} will contain the absolute value of the \textit{largest} change at any node during that single iteration. If \texttt{maxch} is less than the prespecified convergence criterion, stop the iteration; otherwise conduct one more iteration.

15.24. Consider the problem defined by the two-dimensional Poisson equation

\[ \frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} = -(1 - x^2)(1 - y^2) \]

to be solved in the square region \( R \) defined by \(-1 \leq x, y \leq 1\) subject to the Dirichlet boundary conditions requiring \( U = 0 \) on the entire boundary of \( R \). Assume all variables are dimensionless. (a) Obtain a surface plot of this inhomogeneity over the \( x, y \) plane. (b) Recasting \textit{fem2d} to incorporate

\[^{82}\alpha \geq 2 \] generates an unstable algorithm.
the given boundary conditions and to address the non-zero inhomogeneity find, explore, and (using surface and contour plots as you deem appropriate) display the solution $U(x, y)$ in the square region $R$. Compare the resulting equation with Eq. (15.69) as a guide to interpreting the present problem as a modification of the problem addressed by fem2d. (c) Find the two-dimensional vector field $\mathbf{V} = -\text{grad} U = -\nabla U$ in $R$ and display that field graphically in whatever ways seem appropriate.

15.25. (a) Starting with the Taylor series

$$\varphi(x + \Delta x) = \varphi(x) + \Delta x \varphi'(x) + O(\Delta x^2)$$

evaluate

$$\varphi(x + \Delta x) - \varphi(x)$$

to show that

$$\varphi'(x) = \frac{\varphi(x + \Delta x) - \varphi(x)}{\Delta x} + O(\Delta x)$$

That is, derive the forward difference formula in Eq. (15.74). (b) Deduce the more accurate forward difference formula

$$\varphi'(x) = -\frac{\varphi(x + 2 \Delta x) + 4 \varphi(x + \Delta x) - 3 \varphi(x)}{2 \Delta x} + O(\Delta x^2)$$

Hint: Start by using the Taylor expansion

$$f(x + h) = f(x) + hf'(x) + \frac{1}{2} h^2 f''(x) + O(h^3)$$

to expand $\varphi(x + \Delta x)$ and $\varphi(x + 2 \Delta x)$.

15.26. Starting with Eqs. (15.85) and (15.86), derive Eqs. (15.87), (15.88), and (15.89).

15.27. Following the pattern illustrated in Section 15.16.1, develop equations corresponding to Eqs. (15.145)–(15.148) to solve the wave equation

$$\frac{1}{c^2} \frac{\partial^2 u(x, t)}{\partial t^2} = \frac{\partial^2 u(x, t)}{\partial x^2}$$

subject to the boundary and initial conditions

$$u(0, t) = 0 ; \quad u(L, t) = 0 ; \quad u(x, 0) = f(x) ; \quad \left. \frac{\partial u(x, t)}{\partial t} \right|_{t=0} = 0$$

These equations describe a string of length $L$ that is fixed at both ends and set into motion by drawing it aside to the initial shape given by $f(x)$ and released from rest. Of course, for consistency, $f(0)$ and $f(L)$ must both be zero, but $f(x)$ is otherwise unconstrained. Then, develop a program analogous to fdwave1d to solve this problem, testing your program with $f(x) = A \sin \frac{n \pi x}{L}$ for $n = 1, 2, 3$. You should recognize that this exercise asks you to find the lowest three normal modes in the vibrations of this string. You should, of course, expect the results of your program to reveal what you know to be the periodic motion in these three modes of oscillation of the string. Your program should also allow you to confirm that the frequencies of the three modes $f_1$, $f_2$, and $f_3$ are related by $f_2 = 2f_1$ and $f_3 = 3f_1$. Agreement with these expectations provides some level of confidence in the adequacy of your choice of parameters.
/* PROGRAM fdm1dla.c */

/* Note that, when this program has completed executing, 
the file fdm1dla_c.dat containing a first row with the value 
of N+1 followed by N+1 rows, the I-th of which contains 
the I-th value of x in x[I] and the I-th value of the solution 
in inhomogeneous[I], will exist in the default directory. */

#include <stdio.h>
#include <math.h>
#include <stdlib.h> /* needed for the function malloc */

#include <stdlib.h>

void main()
{
    FILE *fptr;
    float alpha, beta, f, L, p, gamma, q;
    int I, N, NN, info;
    float *x; /* An asterisk before a variable declares */
    float *inhomo; /* declres that variable to be a pointer */
    float *diag;
    float *supdiag;
    float *subdiag;
    float dx, dx2;

    printf( "Enter number of segments (N): " ) ; scanf( "%d", &N );
    printf( "Enter alpha: " ) ; scanf( "%f", &alpha );
    printf( "Enter beta: " ) ; scanf( "%f", &beta );
    printf( "Enter f: " ) ; scanf( "%f", &f );
    printf( "Enter L: " ) ; scanf( "%f", &L );
    printf( "Enter p: " ) ; scanf( "%f", &p );
    printf( "Enter gamma: " ) ; scanf( "%f", &gamma );
    printf( "Enter q: " ) ; scanf( "%f", &q );

    x = (float*)malloc( (N+1)*sizeof(float) );
    inhomogeneous = (float*)malloc( (N+1)*sizeof(float) );
    diag = (float*)malloc( (N+1)*sizeof(float) );
    supdiag = (float*)malloc( N*sizeof(float) );
    subdiag = (float*)malloc( N*sizeof(float) );

    dx = L/N; dx2 = pow( dx, 2 );
    for( I=0; I<=N; I++ ) x[I] = dx*I;

    for( I=0; I<=N; I++ ) {
        inhomogeneous[I] = f*dx2;
        diag[I] = 2.0*alpha + beta*dx2;
    };
    inhomogeneous[0] = p;
    inhomogeneous[N] = inhomogeneous[N] + 2.0*q*dx;
    diag[0] = 1.0;
diag[N] = diag[N] + 2.0*gamma*dx;

for( I=0; I<=N-1; I++ ) {
    supdiag[I] = -alpha;
    subdiag[I] = -alpha;
}

supdiag[0] = 0.0;
subdiag[N-1] = -2.0*alpha;

NN = N + 1; I = 1;
sgtsv_( &NN, &I, subdiag, diag, supdiag, inhomo, &NN, &info );

fptr = fopen( "fdm1dla_c.dat", "w" );
fprintf( fptr, "%d
", N + 1 );
for( I=0; I<=N; I++ ) fprintf( fptr, "%.7f %.10f\n", x[I], inhomo[I] );
fclose( fptr );

15.B Evaluating Integrals for 1D FEM Problem

15.B.1 ... using MAXIMA

An appropriate batch file to use MAXIMA to evaluate the important integrals in Section 15.1.7 is named FEM1DCalcs.mac, can be copied from the directory $HEAD/maxima, and invoked within MAXIMA with the statement batch("FEM1DCalcs.mac") issued at MAXIMA’s prompt for commands. The batch file is listed (without output) in the remainder of this subsection.

/* FEM1DCalcs.mac */

/* This batch file evaluates the several integrals that appear in */
/* setting up coding to solve the 1D FEM problem in CPSUP. */

/* Create interpolation functions and their derivatives */

n : [(x2 - x)/l, (x - x1)/l ]$
dnx : diff( n, x )$

/* Create integrand for K, then integrate */

K1 : transpose(dnx) . dnx$
K2 : transpose(n) . n$
K : alpha*K1 + beta*K2$
K : integrate( K, x, x1, x2 )$
K : expand( subst( x1+1, x2, K ) );

/* Evaluate b */

b : integrate( f*n, x, x1, x2 )$
b : expand( subst( x1+1, x2, b ) );
15.B.2 ... using MAPLE

An appropriate batch file to use MAPLE to evaluate the important integrals in Section 15.1.7 is named FEM1DCalcs.mpl, can be copied from the directory $HEAD/maple, and invoked within MAPLE with the statement read "FEM1DCalcs.mpl"; issued at MAPLE's prompt for commands. The batch file is listed (without output) in the remainder of this subsection.

```plaintext
# FEM1DCalcs.mpl
# This batch file evaluates the several integrals that appear in
# setting up coding to solve the 1D FEM problem in CPSUP.

with( linalg ):
interface( echo=2 ):

# Create interpolation functions and their derivatives
n := [ (x2 - x)/l, (x - x1)/l ]:
dnx := diff( n, x ):

# Create integrand for K, then integrate
K1 := multiply( dnx, transpose(dnx) ):
K2 := multiply( n, transpose(n) ):
K := matadd( K1, K2, alpha, beta ):
K := map( integrate, K, x=x1..x2 ):
K := simplify( eval( K, x2=x1+l ) );

# Evaluate b
b := map( integrate, scalarmul( n, f ), x=x1..x2 ):
b := simplify( eval( b, x2 = x1 + l ) );
```

15.B.3 ... using Mathematica

An appropriate batch file to use Mathematica to evaluate the important integrals in Section 15.1.7 is named FEM1DCalcs.mtm, can be copied from the directory $HEAD/mathematica, and invoked within Mathematica with the statement Get[ "FEM1DCalcs.mtm"] issued at Mathematica's prompt for commands. The batch file is listed (without output) in the remainder of this subsection.

```plaintext
(* FEM1DCalcs.mtm *)

(* This batch file evaluates the several integrals that appear in *)
(* setting up coding to solve the 1D FEM problem in CPSUP. *)

(* Create interpolation functions and their derivatives *)
n[1] = (x2-x)/l; n[2] = (x-x1)/l ;
dnx[1] = D[ n[1], x ]; dnx[2] = D[ n[2], x ];

(* Create integrand for K, then integrate and display *)
K1 = Table[ dnx[i]*dnx[j], {i, 2}, {j, 2} ];
```
K2 = Table[n[i]*n[j], {i, 2}, {j, 2}];
K = a*K1 + b*K2;
K = Integrate[K, {x, x1, x2}];
K = Simplify[K /. x2 -> x1 + l];
Print["K =", K // MatrixForm]

(* Evaluate and display b *)

b = Integrate[f*[n[1], n[2]], {x, x1, x2}];
b = Simplify[b /. x2 -> x1 + l];
Print["b = ", b]

15.C Program(s) for FEM Approach to 1D Problem

15.C.6 Listing of fem1dla.c (C)

/* PROGRAM fem1dla.c */

#include <stdio.h>
#include <math.h>
#define M 20

void main()
{
    float alpha, beta, f, l, p, gamma, q, K[M+1][M+1];
    float s, s2, T, u, B[M+1][1], u2, IPIV[M+1];
    int MM, i, j, INFO;

    printf("Enter alpha: "); scanf("%f", &alpha);
    printf("Enter beta: "); scanf("%f", &beta);
    printf("Enter f: "); scanf("%f", &f);
    printf("Enter l: "); scanf("%f", &l);
    printf("Enter p: "); scanf("%f", &p);
    printf("Enter gamma: "); scanf("%f", &gamma);
    printf("Enter q: "); scanf("%f", &q);

    for( i=0; i<=M; i++)
    {
        B[i][0] = 0.0; IPIV[i]=0;
        for(j=0; j<=M; j++) K[i][j] = 0.0;
    }

    s = alpha/l + beta*l/3.0; /* Evaluate common quantities */
    s2 = 2.0*s;
    T = -alpha/l + beta*l/6.0;
    K[0][0] = s; /* Set diagonal elements of K */
    for(i=1; i<M; i++) K[i][i]=s2;
    K[M][M] = s;

    for(i=0; i<=(M-1); i++) /* Set elements above and below */
    {
        K[i+1][i] = T; /* main diagonal of K */
        K[i][i+1] = T;
    }

    for( i=0; i<=M; i++)
    {
        for(j=0; j<=i; j++)
        {
            if(i==j)
            {
                K[i][j] = 10.0;
            }
            else
            {
                K[i][j] = 0.0;
            }
        }
    }

    for( i=1; i<=M; i++)
    {
        float x = (i-1)*l/M;
        float y = x2*(x2-x);
        float z = x2*(x-x1);
        float w = x2*(x2-x1);
        float u = x2*(x2-x1);
15.D C Programs for FDM Approach to 2D Problems

15.D.1 Listing of fdmwave1d.c

/* PROGRAM fdmwave1d.c */

#include <stdio.h> /* For I/O */
#include <math.h> /* For math functions */
#include <stdlib.h> /* To define exit, malloc */
#define pi 3.1415926535

void main()
{
    /* Declare needed variables */

    float *x, *u1, *u2, *u3; /* Pointers to solutions */
    int i, j; /* For loop counters */
    float dt, c, l, dx, alpha, b, t; /* For parameters */
    int n, nt, nf; /* For parameters */
    FILE *fpotr; /* For file pointer */

    /* Request input of necessary parameters, assuring that each
    is stored with the proper data type, calculate dx and
    alpha, and allocate memory for several vectors */

    u = f*l/2.0; u2 = 2.0*u; /* Evaluate common quantities */
    B[0][0] = u; /* Set elements of B */
    for(i=1; i<M; i++) B[i][0] = u2;
    B[M][0] = u;

    K[M][M] = K[M][M] + gamma;
    B[M][0] = B[M][0] + q;
    K[0][0] = 1.0;
    B[0][0] = p;
    for(j=1; j<=M; j++) K[j][0] = 0.0;
    for(i=1;i<=M;i++) B[i][0] = B[i][0] - K[0][i]*p;
    for(i=1; i<=M; i++) K[0][i] = 0.0;
    i = 1; MM = M+1;
    sgesv_(&MM, &i, &K, &MM, &IPIV, &B, &MM, &INFO);

    FILE *fpotr;
    fpotr = fopen( "fem1dla_c.dat", "w" );
    fprintf( fpotr, "%d\n",M+1);
    for( i=0; i<=M; i++) fprintf( fpotr, "%7.2f %10.4f\n", l*i, B[i][0] );
    fclose( fpotr );
}
printf( "Number of segments: "); scanf ( "%d", &n );
printf( "Time step: "); scanf ( "%f", &dt );
printf( "Number of time steps: "); scanf ( "%d", &nt );
printf( "Speed of propagation: "); scanf ( "%f", &c );
printf( "Length of string: "); scanf ( "%f", &l );
printf( "Plot frequency: "); scanf ( "%d", &nf );

dx = l/n;
alpha = pow(c,2)*pow(dt,2)/pow(dx,2);

x = (float*)malloc((n+1)*sizeof(float));
u1 = (float*)malloc((n+1)*sizeof(float));
u2 = (float*)malloc((n+1)*sizeof(float));
u3 = (float*)malloc((n+1)*sizeof(float));

/* Display alpha and trap values that are too large */

printf( " alpha = %8.3f\n", alpha );
if (alpha > 1.0){
    printf("\n alpha > 1; execution halted\n ");
    exit(0);
}

/* Initialize u1, u2, and u3 and set values of x along string */

for(i=0; i<=n; i++){
u1[i]=0.0; u2[i]=0.0; u3[i]=0.0;
x[i] = dx*i;
}

/* Open file and write parameters to it */

fptr = fopen( "fdmwave1d_c.dat", "w" );
fprintf( fptr, "%d %f %f %f \n", n, c, l, alpha );

/* Initialize and display t, u1, and u2 */

t = 0.0;
b = 8.0*pi/l;
for(i = 3*n/8 + 1; i<=5*n/8; i++)
    u1[i] = 1.0 + cos(b*(x[i]-l/2.0));

fprintf( fptr, "%7.3f\n", t );
for( i=0; i<=n; i++)
    fprintf( fptr, "%7.3f", u1[i] );
fprintf( fptr, \n" );

for(i=0; i<=n; i++)
    u2[i]=u1[i];
t = t + dt;
if (nf == 1){
    fprintf( fptr, "%7.3f\n", t );
    for( i=0; i<=n; i++)
        fprintf(fptr, "%7.3f", u2[i]);
    fprintf( fptr, \n" );
/* Evaluate and display time and solution at requested frequency */
for(j=2; j<=nt; j++)
{ u3[0]=0.0;
  for(i=1; i<n; i++)
    u3[i] = alpha*u2[i+1] + 2.0*(1.0-alpha)*u2[i] + alpha*u2[i-1] - u1[i];
  u3[n] = 2.0*alpha*u2[n-1]+2.0*(1.0-alpha)*u2[n] - u1[n];
  t = dt*j;
  if (nf*(j/nf) == j)
  { fprintf( fptr, "%7.3f\n", t );
    for( i=0; i<=n; i++)
      fprintf(fptr, "%7.3f", u3[i]);
    fprintf( fptr, "\n" );
  }
  for(i=0; i<=n; i++)
  { u1[i] = u2[i]; u2[i] = u3[i]; }
}
fclose( fptr );

15.D.2  Listing of fdmdiffus1d.c

/* Command file fdmdiffus1d.c */
#include <stdio.h>    /* For I/O */
#include <math.h>     /* For math functions */
#include <stdlib.h>   /* To define exit() */
#define pi 3.1415926535

void main()
{ /* Declare needed variables */
  float *x, *u1, *u2;     /* Pointers for solutions */
  int i, j;                /* For loop counter */
  float dt, alpha, l, dx, ggamma, t, b; /* For parameters */
  int n, nt, nf;         /* For parameters */
  FILE *fptr;             /* For file pointer */

  /* Request input of necessary parameters, assuring that each is
   stored with the proper data type, and calculate dx and gamma */
  printf( "\nNumber of segments: "); scanf ( "%d", &n );
  printf( "Time step: "); scanf ( "%f", &dt );
  printf( "Number of time steps: "); scanf ( "%d", &nt );
  printf( "Value of alpha: "); scanf ( "%f", &alpha );
  printf( "Length of rod: "); scanf ( "%f", &l );
  printf( "Plot frequency: "); scanf ( "%d", &nf );
  dx = l/n;
  ggamma = pow(alpha,2)*dt/pow(dx,2);

  /* Display ggamma and trap values that are too large */
printf( ' gamma = %8.3f\n', ggamma );
if (ggamma > 0.5)
{ printf("\n gamma > 0.5; execution halted\n" );
 exit(0);
}

/* Allocate memory and initialize u1 and u2, 
 and set values of x along the rod */
x = (float*)malloc((n+1)*sizeof(float));
u1 = (float*)malloc((n+1)*sizeof(float));
u2 = (float*)malloc((n+1)*sizeof(float));

for(i=0; i<=n; i++)
{ u1[i]=0.0; u2[i]=0.0;
 x[i] = dx*i;
}

/* Open file, display parameters, initialize and display t and u1 */
fptr = fopen( "fdmdiffus1d_c.dat", "w" );
fprintf( fptr, "%d %f %f %f \n", n, alpha, l, ggamma );

for(i = 3*n/8 + 1; i<=5*n/8; i++)
 u1[i] = 1.0 + cos(b*(x[i]-l/2.0));
fprintf( fptr, "%7.3f\n", t );
for( i=0; i<=n; i++)
 fprintf( fptr, "%8.4f", u1[i] );
 fprintf( fptr, \n" );

/* Evaluate and display time and temperature distribution */

for(j=1; j<=nt; j++)
{ u2[0]=0.0;
 for(i=1; i<n; i++)
 u2[i] = ggamma*u1[i-1] + (1.0-2.0*ggamma)*u1[i] + ggamma*u1[i+1];
 u2[n] = 2*ggamma*u1[n-1] + (1.0-2.0*ggamma)*u1[n];
 if (nf*(j/nf) == j)
 { fprintf( fptr, "%7.3f\n", j*dt );
  for( i=0; i<n; i++)
   fprintf( fptr, "%8.4f", u2[i] );
  fprintf( fptr, \n" );
 }
 for(i=0; i<=n; i++)
 u1[i] = u2[i];
}
fclose(fptr);
15.D.3 Listing of fdmlap2d.c

/* Command file fdmlap2d.c */

#include <stdio.h>     /* For I/O */
#include <math.h>      /* For math functions */
#define ln 101           /* Use parameter for maximum dimension */
                       /* to facilitate editing */

void main()
{
    /* Declare needed variables */

    float x[ln], y[ln], u[ln][ln]; /* For solutions */
    float l, dx;                     /* For parameters */
    int i, j, itcnt;                /* For loop counters */
    int n, nf;                      /* For parameters */
    int maxits;                     /* Maximum number of iterations */

    /* Request input of necessary parameters, assuring that each is
    stored with the proper data type, calculate dx, and
    initialize x, y, and u */

    printf( "\nNumber of segments: "); scanf( "%d", &n );
    printf( "Length of side: "); scanf( "%f", &l );
    printf( "Maximum number of iterations: "); scanf("%d", &maxits);
    printf( "Display frequency: "); scanf( "%d", &nf );
    itcnt = 0;
    dx = l/n;
    for(i=0; i<n; i++)
    {
        x[i]=dx*i; y[i]=dx*i;
        for(j=0; j<n; j++)
            u[i][j] = 0.0;
    }

    /* Set Dirichlet boundary conditions */
    for(i=0; i<n; i++)
    {
        u[i][0] = 100.0-100.0*x[i]/l; u[0][i] = 100.0;
    }

    /* Display initial guess, labeling it as Iterate 0 */

    printf(" Iterate 0\n" );
    for(i=0; i<n; i++)
    {
        for(j=0; j<n; j++)
        {
            printf( "%10.4f", u[i][j] );
            printf( "\n" );
        }
    }

    /* Evaluate and display iterate number and solution at the iterate */
    for(itcnt=1; itcnt<=maxits; itcnt++)
    {
        for(i=1; i<n; i++)
15.D.4 Listing of fdmlap2d_file.c

/* Command file fdmlap2d_file.c */

#include <stdio.h>    /* For I/O */
#include <math.h>     /* For math functions */
#define ln 101          /* Use parameter for maximum dimension */
                      /* to facilitate editing */

void main()
{ /* Declare needed variables */

    float x[ln], y[ln], u[ln][ln];    /* For solutions */
    float l, dx;                      /* For parameters */
    int i, j, itcnt;                  /* For loop counters */
    int n, nf;                        /* For parameters */
    int maxits;                       /* Maximum number of iterations */
    FILE *fptr;                        /* For file pointer */

    /* Request input of necessary parameters, assuring that each is 
    stored with the proper data type, calculate dx, and 
    initialize x, y, and u */

    printf( "\nNumber of segments: "); scanf( "%d", &n );
    printf( "Length of side: "); scanf( "%.1f", &l );
    printf( "Maximum number of iterations: "); scanf("%d", &maxits);
    printf( "Display frequency: "); scanf( "%.1d", &nf );
    itcnt = 0;
    dx = l/n;
    for(i=0; i<n; i++)
    { x[i]=dx*i; y[i]=dx*i;
        for(j=0; j<n; j++)
            u[i][j] = 0.0;
    }

    /* Open file and write parameters to file */

In this appendix, we deduce a two-dimensional analog to the familiar one-dimensional formula for integration by parts. Consider the integral

\[ I = \int \int_{\Omega} \psi \left( \frac{\partial V_x}{\partial x} + \frac{\partial V_y}{\partial y} \right) \, dx \, dy \]  

(15.225)
where \( \psi, V_x, \) and \( V_y \) are functions of \( x \) and \( y \), and \( \Omega \) is the region in the \( xy \) plane over which the 2D integral is to be evaluated. Recast the integral in the form

\[
I = \int \int_{\Omega} \left( \frac{\partial(\psi V_y)}{\partial x} + \frac{\partial(\psi V_x)}{\partial y} \right) \, dx \, dy - \int \int_{\Omega} \left( V_x \frac{\partial \psi}{\partial x} + V_y \frac{\partial \psi}{\partial y} \right) \, dx \, dy
\]  

(15.226)

Now, introduce the vector \( \mathbf{Q} \) whose \( x \) and \( y \) components are \(-\psi V_y\) and \(\psi V_x\), respectively. Since

\[
\frac{\partial(\psi V_y)}{\partial x} + \frac{\partial(\psi V_x)}{\partial y} = \frac{\partial Q_y}{\partial x} - \frac{\partial Q_x}{\partial y} = (\nabla \times \mathbf{Q})_z = (\nabla \times \mathbf{Q}) \cdot \hat{k}
\]

(15.227)
the integral of concern then can be written in the form

\[
I = \int \int_{\Omega} (\nabla \times \mathbf{Q}) \cdot \hat{k} \, dx \, dy - \int \int_{\Omega} \left( V_x \frac{\partial \psi}{\partial x} + V_y \frac{\partial \psi}{\partial y} \right) \, dx \, dy
\]

(15.228)

which, upon invoking Stokes’ theorem, we can rewrite further in the form

\[
I = \oint_{\Gamma} \mathbf{Q} \cdot \hat{t} \, dl - \int \int_{\Omega} \left( V_x \frac{\partial \psi}{\partial x} + V_y \frac{\partial \psi}{\partial y} \right) \, dx \, dy
\]

(15.229)
where \( \Gamma \) is the path in the \( xy \) plane bounding the region \( \Omega \) and \( \hat{t} \) is a unit vector tangent to that path and pointing in the direction of the thumb of the right hand when the fingers grasp the path while piercing the region \( \Omega \) in the \( z \) direction. The one-dimensional integral in this result is the analog of the unintegrated term in the more familiar one-dimensional formula for integration by parts.

That one dimensional integral involves the component of the vector \( \mathbf{Q} \) tangent to the path \( \Gamma \). We are actually better served by casting things in terms of components normal to that curve. Thus, we introduce the unit vector \( \hat{n} \) normal to the curve and pointing outward from the perspective of a viewer in the region \( \Omega \). Since \( \hat{t} = \hat{k} \times \hat{n} \), we find that

\[
\hat{t} = \hat{k} \times \hat{n} = \hat{k} \times (n_x \hat{i} + n_y \hat{j}) = -n_y \hat{i} + n_x \hat{j}
\]

(15.230)

Thus,

\[
\mathbf{Q} \cdot \hat{t} = n_x Q_y - n_y Q_x = n_x \psi V_x + n_y \psi V_y = (\psi V_x \hat{i} + \psi V_y \hat{j}) \cdot \hat{n} = \psi(V_x \hat{i} + V_y \hat{j}) \cdot \hat{n}
\]

(15.231)
In other words, the tangential component of \( \mathbf{Q} \) can be recast as the normal component of the vector from which \( \mathbf{Q} \) was originally derived. Thus, we find that the original integral in Eq. (15.225) can alternatively be evaluated as

\[
I = -\int \int_{\Omega} \left( V_x \frac{\partial \psi}{\partial x} + V_y \frac{\partial \psi}{\partial y} \right) \, dx \, dy + \oint_{\Gamma} (\psi V_x \hat{i} + \psi V_y \hat{j}) \cdot \hat{n} \, dl
\]

(15.232)

### 15.F Evaluating Integrals for 2D FEM Problem

A symbol manipulating program can usefully be invoked to evaluate the several integrals that appear in the solution of the example problem introduced in Section 15.23.

#### 15.F.1 . . . using MAXIMA

An appropriate batch file to use MAXIMA to evaluate the necessary integrals when array indices start at 0 is named FEM2DCalcs0.mac, can be copied from the directory \$HEAD/maxima and, with the directory containing this file in MAXIMA’s search path,\(^{83}\) invoked within MAXIMA with the statement `batch( "FEM2DCalcs0.mac" )` issued at MAXIMA’s prompt for commands. This batch file includes the statements

\(^{83}\)Alternatively, the full path to the file can be included within the quotation marks in the indicated command.
/* FEM2DCalcs0.mac */

/* This batch file evaluates several integrals that appear in */
/* setting up coding to solve the 2D FEM problem in CPSUP */
/* for programs whose array indices start at 0. */

/* Create interpolation functions and their derivatives */

r : [1, x, y]$ /* Create rows */
r0 : [1,x0,y0]$ 
r1 : [1,x1,y1]$ 
r2 : [1,x2,y2]$

delta : determinant( matrix( r0,r1,r2 ) )$ /* Evaluate denominator */

n0 : determinant( matrix( r,r1,r2 ) )/delta$ /* Evaluate functions */
n1 : determinant( matrix( r0,r,r2 ) )/delta$
n2 : determinant( matrix( r0,r1,r ) )/delta$
n : [ n0, n1, n2 ]$

/* Simplify functions and evaluate derivatives */
/* for odd-numbered elements */

(x1 : x0 + dx, x2 : x0 + dx, y1 : y0, y2 : y0 + dx )$
nodd : expand( ev(n) )$
dnoddx : diff( nodd, x )$
dnoddy : diff( nodd, y )$

/* Simplify functions and evaluate derivatives */
/* for even-numbered elements */

(x1 : x0 + dx, x2 : x0, y1 : y0 + dx, y2 : y0 + dx )$
neven : expand( ev(n) )$
dnevenx : diff( neven, x )$
dneveny : diff( neven, y )$

remvalue( x1, x2, y1, y2 )$

/* Create integrand for Kodd, then integrate */

Kodd : ax*transpose(dnoddx).dnoddx + ay*transpose(dnoddy).dnoddy + b*transpose(nodd).nodd$
Kodd : integrate( integrate(Kodd, y, y0, y0+x-x0 ), x, x0, x0+dx )$
Kodd : expand( Kodd )$

/* Create integrand for Keven, then integrate */

Keven : ax*transpose(dnevenx).dnevenx + ay*transpose(dneveny).dneveny + b*transpose(neven).neven$
Keven : integrate( integrate(Keven, y, y0+x-x0, y0+dx ), x, x0, x0+dx )$
Keven : expand( Keven )$

/* Evaluate bodd and beven */
bodd : integrate( integrate( nodd*f, y, y0, y0+x-x0 ), x, x0, x0+dx )$

beven : integrate( integrate( neven*f, y, y0+x-x0, y0+dx ), x, x0, x0+dx )$

/* Evaluate needed g’s */

g : integrate( nodd*q, y, y0, y0+dx )$
g : expand( subst( x0+dx, x, g ) );

An appropriate batch file to use MAXIMA to evaluate the necessary integrals when array indices start at 1 is named FEM2DCalcs1.mac, can be copied from the directory "$HEAD/maxima" and, with the directory containing this file in MAXIMA’s search path, invited within MAXIMA with the statement batch( "FEM2DCalcs1.mac" ) issued at MAXIMA’s prompt for commands. This batch file includes the statements

/* FEM2DCalcs1.mac */

/* This batch file evaluates several integrals that appear in */
/* setting up coding to solve the 2D FEM problem in CPSUP */
/* for programs whose array indices start at 1. */

/* Create interpolation functions and their derivatives */

r : [1, x, y]$
r1 : [1,x1,y1]$
r2 : [1,x2,y2]$
r3 : [1,x3,y3]$

delta : determinant( matrix( r1,r2,r3 ) )$

n1 : determinant( matrix( r,r2,r3 ) )/delta$

n2 : determinant( matrix( r1,r,r3 ) )/delta$

n3 : determinant( matrix( r1,r2,r ) )/delta$

n : [ n1, n2, n3 ]$

/* Simplify functions and evaluate derivatives */
/* for even-numbered elements */

(x2 : x1 + dx, x3 : x1 + dx, y2 : y1, y3 : y1 + dx )$

neven : expand( ev(n) )$

dnevenx : diff( neven, x )$

dneveny : diff( neven, y )$

/* Simplify functions and evaluate derivatives */
/* for odd-numbered elements */

(x2 : x1 + dx, x3 : x1, y2 : y1 + dx, y3 : y1 + dx )$
nodd : expand( ev(n) )$

dnoddx : diff( nodd, x )$

dnodd : diff( nodd, y )$

84 Alternatively, the full path to the file can be included within the quotation marks in the indicated command.
remvalue( x1, x2, y1, y2 )

/* Create integrand for Keven, then integrate */

Keven : ax*transpose(dnevenx).dnevenx + ay*transpose(dneveny).dneveny +
        b*transpose(neven).neven$
Keven : integrate( integrate(Keven, y, y1, y1+x-x1 ), x, x1, x1+dx )$
Keven : expand( Keven );

/* Create integrand for Kodd, then integrate */

Kodd : ax*transpose(dnoddx).dnoddx + ay*transpose(dnoddy).dnoddy +
       b*transpose(nodd).nodd$
Kodd : integrate( integrate(Kodd, y, y1+x-x1, y1+dx ), x, x1, x1+dx )$
Kodd : expand( Kodd );

/* Evaluate bodd and beven */

beven : integrate( integrate( neven*f, y, y1, y1+x-x1 ), x, x1, x1+dx )$
beven : expand( beven );

bodd : integrate( integrate( nodd*f, y, y1+x-x1, y1+dx ), x, x1, x1+dx )$
bodd : expand( bodd );

/* Evaluate needed g's */

g : integrate( neven*q, y, y1, y1+dx )$
g : expand( subst( x1+dx, x, g ) );

15.F.2 ... using MAPLE

An appropriate batch file to use MAPLE to evaluate the necessary integrals when array indices
start at 1 is named FEM2DCalcs0.mpl, can be copied from the directory $HEAD/maple and, with the
directory containing this file in MAPLE's search path, invoked within MAPLE with the statement
read "FEM2DCalcs0.mpl"; issued at MAPLE's prompt for commands. The batch file includes the
statements

# FEM2DCalcs0.mpl

# This batch file evaluates the several integrals that appear in
# setting up coding to solve the 2D FEM problem in CPSUP for
# programs whose array indices start at 0.

with( LinearAlgebra );

# Create interpolation functions and their derivatives

r := [ 1, x, y ]:
r0 := [ 1, x0, y0 ]:
r1 := [ 1, x1, y1 ]:
r2 := [ 1, x2, y2 ]:
Delta := Determinant( matrix( [r0, r1, r2 ] ) ):

n0 := Determinant( matrix( [r, r1, r2] ) ) / Delta: # Evaluate functions
n1 := Determinant( matrix( [r0, r, r2] ) ) / Delta:

n2 := Determinant( matrix( [r0, r1, r] ) ) / Delta:

n := [n0, n1, n2]:

# Simplify functions and evaluate derivatives
# for odd-numbered elements

sb := {x1 = x0 + dx, x2 = x0 + dx, y1 = y0, y2 = y0 + dx}:
nodd := convert( simplify( eval( n, sb ) ), Vector):
dnoddx := map(diff, nodd, x):
dnoddy := map(diff, nodd, y):

# Simplify functions and evaluate derivatives
# for even-numbered elements

sb := {x1 = x0 + dx, x2 = x0, y1 = y0 + dx, y2 = y0 + dx}:
neven := convert(simplify( eval( n, sb ) ), Vector):
dnevenx := map(diff, neven, x):
dneveny := map(diff, neven, y):

# Create integrand for Kodd, then integrate

Kodd1 := Multiply( dnoddx, Transpose(dnoddx) ):
Kodd2 := Multiply( dnoddy, Transpose(dnoddy) ):
Kodd3 := Multiply( nodd, Transpose(nodd) ):
Kodd := Add( Kodd1, Kodd2, ax, ay ):
Kodd := Add( Kodd, Kodd3, 1, b ):
Kodd := map( integrate, Kodd, y=y0..y0+x-x0 ):
Kodd := map( integrate, Kodd, x=x0..x0+dx ):
Kodd := simplify( Kodd );

# Create integrand for Kodd, then integrate

Keven1 := Multiply( dnevenx, Transpose(dnevenx) ):
Keven2 := Multiply( dneveny, Transpose(dneveny) ):
Keven3 := Multiply( neven, Transpose(neven) ):
Keven := Add( Keven1, Keven2, ax, ay ):
Keven := Add( Keven, Keven3, 1, b ):
Keven := map( integrate, Keven, y=y0+dx-x0..y0+x ):
Keven := map( integrate, Keven, x=x0..x0+dx ):
Keven := simplify( Keven );

# Evaluate bodd and beven

bodd := map( integrate, ScalarMultiply(nodd, f), y=y0..y0+x-x0 ):
bodd := map( integrate, bodd, x=x0..x0+dx ):
bodd := simplify( bodd );

beven := map( integrate, ScalarMultiply( neven, f), y=y0+x-x0..y0+dx ):
beven := map( integrate, beven, x=x0..x0+dx ):
beven := simplify( beven );
# Evaluate needed g’s

g := eval( nodd, x=x0+dx );
g := map( integrate, ScalarMultiply(g, q), y=y0..y0+dx );
g := simplify( g );

An appropriate batch file to use MAPLE to evaluate the necessary integrals when array indices
start at 1 is named FEM2DCalcs1.mpl, can be copied from the directory $HEAD/maple, and invoked
within MAPLE with the statement batch( "FEM2DCalcs1.mpl" ) issued at MAPLE’s prompt for
commands. This batch file includes the statements

# FEM2DCalcs1.mpl

# This batch file evaluates the several integrals that appear in
# setting up coding to solve the 2D FEM problem in CPSUP
# for programs whose array indices start at 1.

with( LinearAlgebra );

# Create interpolation functions and their derivatives

r := [ 1, x, y ];
r1 := [ 1, x1, y1 ];
r2 := [ 1, x2, y2 ];
r3 := [ 1, x3, y3 ];

Delta := Determinant( matrix( [r1, r2, r3] ) );

n1 := Determinant( matrix( [r, r2, r3] ) ) / Delta; # Evaluate functions
n2 := Determinant( matrix( [r1, r, r3] ) ) / Delta;
n3 := Determinant( matrix( [r1, r2, r] ) ) / Delta;
n := [ n1, n2, n3 ];

# Simplify functions and evaluate derivatives
# for even-numbered elements

sb := {x2 = x1 + dx, x3 = x1 + dx, y2 = y1, y3 = y1 + dx};
neven := convert(simplify( eval( n, sb ) ), Vector);
dnevenx := map(diff, neven, x);
dneveny := map(diff, neven, y);

# Simplify functions and evaluate derivatives
# for odd-numbered elements

sb := {x2 = x1 + dx, x3 = x1, y2 = y1+dx, y3 = y1 + dx};
nodd := convert(simplify( eval( n, sb ) ), Vector);
dnoddx := map(diff, nodd, x);
dnoddy := map(diff, nodd, y);

# Create integrand for Keven, then integrate

Keven1 := Multiply( dnevenx, Transpose(dnevenx) );
Keven2 := Multiply( dneveny, Transpose(dneveny) );
\[
\text{Keven3} \ := \ \text{Multiply}(\ \text{neven}, \ \text{Transpose}(\text{neven}));
\]
\[
\text{Keven} \ := \ \text{Add}(\ \text{Keven1}, \ \text{Keven2}, \ ax, \ ay);
\]
\[
\text{Keven} \ := \ \text{Add}(\ \text{Keven}, \ \text{Keven3}, \ 1, \ b);
\]
\[
\text{Keven} \ := \ \text{map}(\ \text{integrate}, \ \text{Keven}, \ y=y_1..y_1+x-x_1);
\]
\[
\text{Keven} \ := \ \text{map}(\ \text{integrate}, \ \text{Keven}, \ x=x_1..x_1+dx);
\]
\[
\text{Keven} \ := \ \text{simplify}(\ \text{Keven});
\]

# Create integrand for Kodd, then integrate
\[
\text{Kodd1} \ := \ \text{Multiply}(\ \text{dnoddx}, \ \text{Transpose}(\text{dnoddx}));
\]
\[
\text{Kodd2} \ := \ \text{Multiply}(\ \text{dnoddy}, \ \text{Transpose}(\text{dnoddy}));
\]
\[
\text{Kodd3} \ := \ \text{Multiply}(\ \text{nodd}, \ \text{Transpose}(\text{nodd}));
\]
\[
\text{Kodd} \ := \ \text{Add}(\ \text{Kodd1}, \ \text{Kodd2}, \ ax, \ ay);
\]
\[
\text{Kodd} \ := \ \text{Add}(\ \text{Kodd}, \ \text{Kodd3}, \ 1, \ b);
\]
\[
\text{Kodd} \ := \ \text{map}(\ \text{integrate}, \ \text{Kodd}, \ y=y_1+x-x_1..y_1+dx);
\]
\[
\text{Kodd} \ := \ \text{map}(\ \text{integrate}, \ \text{Kodd}, \ x=x_1..x_1+dx);
\]
\[
\text{Kodd} \ := \ \text{simplify}(\ \text{Kodd});
\]

# Evaluate bodd and beven
\[
\text{beven} \ := \ \text{map}(\ \text{integrate}, \ \text{ScalarMultiply}(\text{neven}, \ f), \ y=y_1..y_1+x-x_1);
\]
\[
\text{beven} \ := \ \text{map}(\ \text{integrate}, \ \text{beven}, \ x=x_1..x_1+dx);
\]
\[
\text{beven} \ := \ \text{simplify}(\ \text{beven});
\]
\[
\text{bodd} \ := \ \text{map}(\ \text{integrate}, \ \text{ScalarMultiply}(\text{nodd}, \ f), \ y=y_1+x-x_1..y_1+dx);
\]
\[
\text{bodd} \ := \ \text{map}(\ \text{integrate}, \ \text{bodd}, \ x=x_1..x_1+dx);
\]
\[
\text{bodd} \ := \ \text{simplify}(\ \text{bodd});
\]

# Evaluate needed g’s
\[
\text{g} \ := \ \text{eval}(\ \text{neven}, \ x=x_1+dx);
\]
\[
\text{g} \ := \ \text{map}(\ \text{integrate}, \ \text{ScalarMultiply}(\text{g}, \ q), \ y=y_1..y_1+dx);
\]
\[
\text{g} \ := \ \text{simplify}(\ \text{g});
\]

15.F.3 ... using Mathematica

An appropriate batch file to use Mathematica to evaluate the necessary integrals when array indices start at 0 is named FEM2DCalcs0.mtm, can be copied from the directory $HEAD/mathematica, and invoked within Mathematica with the statement Get[ "FEM2DCalcs0.mtm"] issued at Mathematica’s prompt for commands. The batch file includes the statements

(* FEM2DCalcs0.mtm *)

(* This batch file evaluates the several integrals that appear in *)
(* setting up coding to solve the 2D FEM problem in CPSUP when *)
(* array indices start at 0. *)

(* Create interpolation functions and their derivatives *)

r = { 1, x, y }
r0 = { 1, x0, y0 }
r1 = { 1, x1, y1 }
r2 = { 1, x2, y2 }
15.F. EVALUATING INTEGRALS FOR 2D FEM PROBLEM

\[ \text{CapDelta} = \det\{r_0, r_1, r_2\} \]
\[ n_0 = \det\{r, r_1, r_2\} / \text{CapDelta} \]
\[ n_1 = \det\{r_0, r, r_2\} / \text{CapDelta} \]
\[ n_2 = \det\{r_0, r_1, r\} / \text{CapDelta} \]
\[ n = \{n_0, n_1, n_2\} \]

(* Simplify functions and evaluate derivatives *)
(* for odd-numbered elements *)
\[ sb = \{x_1 \rightarrow x_0 + dx, x_2 \rightarrow x_0 + dx, y_1 \rightarrow y_0, y_2 \rightarrow y_0 + dx\} \]
\[ nodd = \text{Simplify}\[n /. sb\] \]
\[ dnoddx = \text{D}[nodd, x] \]
\[ dnoddy = \text{D}[nodd, y] \]

(* Simplify functions and evaluate derivatives *)
(* for even-numbered elements *)
\[ sb = \{x_1 \rightarrow x_0 + dx, x_2 \rightarrow x_0, y_1 \rightarrow y_0 + dx, y_2 \rightarrow y_0 + dx\} \]
\[ neven = \text{Simplify}\[n /. sb\] \]
\[ dnevenx = \text{D}[neven, x] \]
\[ dneveny = \text{D}[neven, y] \]

(* Create integrand for Kodd, then integrate *)
\[ \text{Kodd1} = \text{Table}\[dnoddx[i]*dnoddx[j], \{i, 3\}, \{j, 3\}\] \]
\[ \text{Kodd2} = \text{Table}\[dnoddy[i]*dnoddy[j], \{i, 3\}, \{j, 3\}\] \]
\[ \text{Kodd3} = \text{Table}\[nodd[i]*nodd[j], \{i, 3\}, \{j, 3\}\] \]
\[ \text{Kodd} = ax*\text{Kodd1} + ay*\text{Kodd2} + b*\text{Kodd3} \]
\[ \text{Kodd} = \int \text{Kodd}, \{y, y_0, y_0+x-x_0\} \]
\[ \text{Kodd} = \text{Simplify}\[\text{Kodd}\] \]
\[ \text{Print}\[\"Kodd =\", \text{Kodd}//\text{MatrixForm}\] \]

(* Create integrand for Kodd, then integrate *)
\[ \text{Keven1} = \text{Table}\[dnevenx[i]*dnevenx[j], \{i, 3\}, \{j, 3\}\] \]
\[ \text{Keven2} = \text{Table}\[dneveny[i]*dneveny[j], \{i, 3\}, \{j, 3\}\] \]
\[ \text{Keven3} = \text{Table}\[neven[i]*neven[j], \{i, 3\}, \{j, 3\}\] \]
\[ \text{Keven} = ax*\text{Keven1} + ay*\text{Keven2} + b*\text{Keven3} \]
\[ \text{Keven} = \int \text{Keven}, \{y, y_0+x-x_0, y_0+dx\} \]
\[ \text{Keven} = \text{Simplify}\[\text{Keven}\] \]
\[ \text{Print}\[\"Keven =\", \text{Keven}//\text{MatrixForm}\] \]

(* Evaluate bodd and beven *)
\[ \text{bodd} = \int f*nodd, \{y, y_0, y_0+x-x_0\} \]
\[ \text{bodd} = \int \text{bodd}, \{x, x_0, x_0+dx\} \]
\[ \text{bodd} = \text{Simplify}\[\text{bodd}\] \]
\[ \text{Print}\[\"bodd =\", \text{bodd}\]
beven = Integrate[ f*neven, {y, y0+x0-x0, y0+dx} ]
beven = Integrate[ beven, {x, x0, x0+dx} ]
beven = Simplify[ beven ]
Print[ "beven = ", beven ]

(* Evaluate needed g’s *)

g = nodd /. x -> x0+dx
g = Integrate[ q*g, {y, y0, y0+dx} ]
Print[ "g = ", g ]

An appropriate batch file to use Mathematica to evaluate the necessary integrals when array indices start at 1 is named FEM2DCalcs1.mpl, can be copied from the directory $HEAD/mathematica, and invoked within Mathematica with the statement batch("FEM2DCalcs1.mtm") issued at Mathematica’s prompt for commands. This batch file includes the statements
15.G. Program(s) for FEM Approach to 2D Problems

15.G.6 Listing of fem2dla.c (C)

/* PROGRAM fem2dla.c */

#include <stdio.h>
#include <math.h>
#define ND 8 /* Number of segments each side */
#define M ( 2*ND*ND ) /* Determine number of elements */
#define N ( (ND+1)*(ND+1) ) /* Determine number of nodes */

void main()
{
    float AL, ALPHAX, ALPHAY, BETA, F, P1, P2, Q, P3;
    float DX, K[N][N], B[N][1], G[N][1], X[N], Y[N];
    int I, J, NCT, NCM[M][3], NVL, IE, NND;
    float Keven[3][3], Kodd[3][3], BX;
    int L1, L2, INFO, IPIV[N], NU, NS, NT;
    float PHI[N+1][N+1]; /* For solution */
    FILE *fptr; /* For file pointer */

    printf("Enter length of side (L): "); scanf("%f", &AL);
    printf("Enter alpha_x: "); scanf("%f", &ALPHAX);
    printf("Enter alpha_y: "); scanf("%f", &ALPHAY);
    printf("Enter beta: "); scanf("%f", &BETA);
    printf("Enter f: "); scanf("%f", &F);
    printf("Enter value for top edge: "); scanf("%f", &P1);
    printf("Enter value for bottom edge:" ); scanf("%f", &P2);
    printf("Enter q: " ); scanf("%f", &Q);

    DX = AL/ND; /* Calculate segment size */

    for(I=0; I<N; I++) /* Initialize K, B, G */
        for(J=0; J<N; J++) K[I][J] = 0.0;
        B[I][0] = 0.0;
        G[I][0] = 0.0;

    /* Find coordinates of points along axes */

    NCT=0;
    for(I=0; I<ND; I++) /* Start row number loop */
        for(J=0; J<ND; J++) /* Start column number loop */
            { X[NCT] = I*DX; /* Find x coordinate */
                Y[NCT] = AL - J*DX; /* Find y coordinate */
                NCT=NCT+1; /* Increment counter */
            }

    /* Find connectivity matrix */
for(IE=0;IE<M;IE++) /* Loop through all elements */
{
    NVL = IE/(2*ND); /* Find index of vertical line */
    NCM[IE][0] = IE/2 + 1 + NVL; /* Find global number of node 0 */
    if (2*(IE/2) == IE) /* If IE is even */
    {
        NCM[IE][1] = NCM[IE][0] + ND; /* find global number of node 1 */
        NCM[IE][2] = NCM[IE][0] - 1; /* find global number of node 2 */
    }
    else /* If IE is odd */
    {
        NCM[IE][1] = NCM[IE][0] + ND + 1; /* find global number of node 1 */
        NCM[IE][2] = NCM[IE][0] + ND; /* find global number of node 2 */
    }
}

/* Find element contributions to stiffness matrix */
BX = BETA*pow(DX,2); /* Evaluate a common quantity */
Kodd[0][0] = (BX+6*ALPHAX)/12; /* Assign the appropriate value to */
Kodd[1][1] = (BX+6*ALPHAX+6*ALPHAY)/12; /* each K[i,j]. Note that the arrays */
Kodd[2][2] = (BX+6*ALPHAY)/12; /* are symmetric, and that Kodd */
Kodd[0][1] = (BX-12*ALPHAX)/24; /* includes all of the same values as*/
Kodd[1][0] = Kodd[0][1]; /* Kodd, but in different locations. */
Kodd[1][2] = (BX-12*ALPHAY)/24;
Kodd[2][1] = Kodd[1][2];
Kodd[0][2] = BX/24;
Kodd[2][0] = BX/24;
Keven[0][0] = Kodd[2][2];
Keven[1][1] = Kodd[0][0];
Keven[2][2] = Kodd[1][1];
Keven[0][1] = Kodd[0][2];
Keven[1][0] = Kodd[0][1];
Keven[1][2] = Kodd[0][1];
Keven[2][1] = Kodd[1][2];
Keven[0][2] = Kodd[1][2];
Keven[2][0] = Kodd[1][2];

/* Assemble full stiffness matrix */
for(IE=0; IE<M; IE++) /* Count through element numbers */
{
    L1=2*(IE/2); L2=IE; /* L1=L2 if IE is even */
    for(I=0; I<3; I++)
    {
        /* For each local node of element place its contributions at the */
        /* correct locations in K and B */
        B[NCM[IE][1]][0]=B[NCM[IE][1]][0]+F*pow(DX,2)/6;
        for(J=0; J<3; J++)
        {
            if (L1 != L2)
            {
                K[NCM[IE][I]][NCM[IE][J]] = K[NCM[IE][I]][NCM[IE][J]] + Kodd[I][J];
            }
            else
            {
                K[NCM[IE][I]][NCM[IE][J]] = K[NCM[IE][I]][NCM[IE][J]] + Keven[I][J];
            }
        }
    }
}
/* Incorporate boundary conditions */

for(I=0; I<ND+1; I++)
{ 
  NU = I;  /* Nodes on the left boundary */
  NS = I*(ND+1);  /* Nodes on the top boundary */
  NT = (I+1)*(ND+1) - 1;  /* Nodes on the bottom boundary */
  P3 = (P1-P2)/AL * Y[NU] + P2;  /* Find values of phi on left boundary */
  for(J=0; J<N; J++)
  { 
    K[NS][J] = 0.0;  /* Set rows in K to zero where value */
    K[NT][J] = 0.0;  /* of phi is known */
    K[NU][J] = 0.0;
    B[NS][0] = P1;  /* Set values of b */
    B[NT][0] = P2;
    B[NU][0] = P3;
    if(J != NS) B[J][0] = B[J][0] - K[J][NS]*P1;
    if(J != NT) B[J][0] = B[J][0] - K[J][NT]*P2;
    if(J != NU) B[J][0] = B[J][0] - K[J][NU]*P3;
  }
  K[NS][NS] = 1.0;  /* Set the appropriate entry to 1 in */
  K[NT][NT] = 1.0;  /* the rows where phi is known */
  K[NU][NU] = 1.0;
}

/* Calculate G {g} and vector of inhomogenieties */

for(I=0; I<ND; I++)  /* Nodes on right boundary */
{ 
  NND = ND*(ND+1)+1;
  G[NND][0] = Q*DX;
}

for(I=0; I<ND; I++) B[I][0] = B[I][0] + G[I][0];  /* {b} ==> {b+g} */

/* Solve the system of equations */

I=1; J=N;
sgesv_(&J, &I, &K, &J, &IPIV, &B, &J, &INFO);

/* Recast solution as matrix */

NCT=0;  /* Initialize a counter variable */
for(I=0; I<ND; I++)  /* Use nested loops to write all */
{ 
  for(J=0; J<N; J++)  /* entries into PHI */
  { 
    PHI[I][J] = B[NCT][0];
    NCT = NCT + 1;
  }
}

/* output solution to file */

fptr = fopen( "fem2dla_c.dat", "w" );
fprintf( fptr, "%d\n", ND+1 );
for(J=0; J<=ND; J++)
{ for(I=0; I<=ND; I++)
    fprintf(fptr, "%10.4f", PHI[I][J] );
    fprintf( fptr, "\n" );
}
fclose( fptr );
Appendix Z

Contacting Software Vendors

Note: Regardless of which components are included and which omitted in this version of *Computation and Problem Solving in Undergraduate Physics*, the information in this Appendix is that from the assemblage containing all components.

In this appendix, we present information to help interested individuals contact the vendors of software referred to at various points in this book. The information in this appendix was accurate as of 16 April 2018, but no guarantee can be made that it will be accurate forever into the future.

**IDL**

Harris Geospatial Solutions  
385 Interlochen Crescent  
Broomfield, CO 80021 USA  
Voice: 303-786-9900  
FAX: 303-786-9909  
E-mail: geospatial@harris.com  
Web: www.harrisgeospatial.com

In addition, links to numerous third-party contributions of IDL routines can be found by Googling ‘IDL routines’. Among the most prominent of the sites that will emerge points you to the IDL Astronomy User’s Library maintained at the Goddard Space Flight Center and accessible from the URL idlastro.gsfc.nasa.gov.

**LAPACK**

LAPACK is a large FORTRAN 90 package which implements numerous algorithms to accomplish various tasks in linear algebra. Full information is available at www.netlib.org/lapack. The source code is in the public domain. Instructions for downloading the package are included at the referenced website. Those downloads include the routines in BLAS (Basic Linear Algebra Subprograms), which are invoked by routines in LAPACK. The development of the package was supported by the National Science Foundation and the Department of Energy, and maintenance has been supported for many years by MathWorks and Intel. (Netlib is a large repository of programs maintained at Oak Ridge National Laboratory.)

**\LaTeX – see TeX**

**LSODE**

The ODE solver LSODE is one component in a large package of ODE solvers originating in the Computing and Mathematics Research Division of the Lawrence Livermore National Laboratory. The full package is called ODEPACK, public-domain software written in FORTRAN by Alan C. Hindmarsh and others. Information about compiled solvers, numerous example programs, and source code for the package are available for download from links at computing.llnl.gov/projects/odepack and computing.llnl.gov/projects/odepack/software. Additional information and downloads may be found from links at www.netlib.org/odepack. In particular, the two text files opkd-sum and
opks-sum at this URL provide a description of the single- and double-precision components in the package. (Netlib is a large repository of programs maintained at Oak Ridge National Laboratory.)

**Mac\TeX**

Mac\TeX is a shareware implementation of the \TeX/L\A\TeX system for Macintosh computers. Information is available at the URL www.tug.org/mactex.

**MAXIMA**

Originally called MACSYMA, this first of the computer algebra systems was developed at MIT and supported from its origin in 1968 until 1982 by MIT, NASA, ONR, and DOE. In 1982, MACSYMA became a commercial product that was further developed and remained available until about 1999. The 1982 MIT version remained available as DOE MACSYMA but was released in 1999 to a group that continues to develop and maintain the program, changing its name to MAXIMA, whose website is at the URL http://maxima.sourceforge.net/. MAXIMA is freely available for a wide variety of platforms.

**MAPLE®**

Waterloo Maple, Inc.                          Voice:  800-267-6583
615 Kumpf Drive                             FAX:      519-747-5284
Waterloo, Ontario                          E-mail: info@maplesoft.com
CANADA, N2V 1K8                              Web:       www.maplesoft.com

**MATHEMATICA®**

100 Trade Center Drive                    FAX:     217-398-0747
Champaign, IL 61820-7237 USA              E-mail: info@wolfram.com
Web:       www.wolfram.com

**MATLAB®**

The MathWorks, Inc.                            Voice:  508-647-7000
3 Apple Hill Drive                          FAX:      508-647-7101
Natick, MA 01760-2098 USA                   E-mail: info@mathworks.com
Web:       www.mathworks.com

**MiK\TeX**

MiK\TeX is an implementation of the \TeX/L\A\TeX system for Windows computers. Information is available at the URL miktex.org. MiK\TeX is also available from CTAN, the \TeX/L\A\TeX distribution network accessible from links at www.tug.org.

**MUDPACK**

The freely-available package of FORTRAN PDE solvers called MUDPACK, written by John C. Adams and others, originated at the National Center for Atmospheric Research (NCAR). The main web page for information about this FORTRAN package, and many others, is www2.cisl.ucar.edu/research-software/software. MUDPACK can be downloaded from the webpage github.com/NCAR/NCAR-Classic-Libraries-for-Geophysics. While this package is no longer under active development, it remains available and useful.

**Numerical Algorithms Library**

The Numerical Algorithms Library (NAG library) is a large commercially available library of C and Fortran subroutines/subprograms implementing a wide assortment of numerical and statistical algorithms.

Numerical Algorithms Group, Inc.    Voice:  630-971-2337
801 Warrenville Road                 FAX:      630-971-2706
Suite 185                            E-mail: infodesk@nag.com
Lisle, IL 60532-4332 USA             Web:       www.nag.com
NUMERICAL RECIPES

The vendor of Numerical Recipes is reluctant to provide detailed contact information, preferring that potential customers deal with them through forms on their website at www.numerical.recipes (without a ‘.com’). The current version and many of the past versions, some of which include languages no longer being updated, are available for download from this site.

OCTAVE

Available under the terms of the GNU General Public License as published by the Free Software Foundation, OCTAVE is an array processing program whose syntax is similar to that of MATLAB. Information about the program can be found at the site octave.org.

ODEPACK – see LSODE

OzTEX

OzTEX is a shareware implementation of the \TeX/La\TeX{} system for Macintosh computers. Information is available at the URL www.trevorrow.com/oztex. OzTEX is also available from CTAN, the \TeX/La\TeX{} distribution network accessible from links at www.tug.org.

PYTHON/NUMPY/MATPLOTLIB/PLOTLY/MAYAVI/SCIKIT-IMAGE

PYTHON, a high-level programming language created by Guido van Rossum, was first released in 1991. In execution, PYTHON programs are interpreted, not compiled. Information and free downloads are available at the URL www.python.org. Information about various add-on modules can be found as follows:

- The module numpy, which adds numerous mathematical capabilities (arrays, matrices, mathematical functions, ...) to PYTHON, is described at the URL www.numpy.org.
- The module matplotlib, which adds numerous 2D plotting capabilities and a few 3D plotting capabilities to PYTHON, is described at the URL matplotlib.org.
- The module plotly, which provides several 3D plotting capabilities, is described at the URL plot.ly.
- The module mayavi, which provides numerous capabilities for 3D data visualisation and plotting, is described at the URL docs.enthought.com/mayavi/mayavi.
- The module scikit-image, which includes scimage and provides image processing capabilities, is described at the URL scikit-image.org.

There are numerous distributions of PYTHON. A popular distribution that automatically includes numerous modules that might not be included in many other distributions is described at www.anaconda.com.

\TeX/La\TeX/dvips/graphicx/makeindex/xdvi/dvips/

The primary site for information (history, current plans, downloads, ...) for \TeX{}, \LaTeX{}, and numerous other publicly available components of \TeX{} and its derivatives is the web site of the \TeX{} Users’ Group (TUG), www.tug.org. This organization maintains CTAN (the Comprehensive \TeX{} Archive Network), which has a handful of backbone machines around the world and a number of mirror sites, from any of which an enormous number of files associated with the \TeX/La\TeX{} system can be downloaded.

\TeX{}live

\TeX{}live is among the newer implementations of the \TeX/La\TeX{} system for all platforms. Information about this distribution and instructions for downloading and installing it are available from the URL www.tug.org/texlive.
TGIF

TGIF (pronounced T-G-I-F) is a versatile program for creating two-dimensional drawings. The program is Xlib-based and interactive, and it runs under X11 on LINUX and UNIX platforms (including MAC OS X and cygwin on Windows). Information (brief history, licensing understandings, instructions for downloading, ...) about TGIF is available from the URL \url{bourbon.usc.edu/tgif/}.

Winedt

Winedt is an inexpensive ($40.00 per student user, $60 per educational user) editor for Intel-based machines running one or another version of Microsoft Windows. It has particular features that make it especially suitable for creating source files for \LaTeX. Information about licensing and downloads can be found at the URL \url{www.winedt.com}. Winedt is also available from CTAN, the \LaTeX distribution network accessible from links at \url{www.tug.org}.

Xemacs

This flexible, open source text editor is available from a variety of sources and is protected under the terms of the GNU Public License. Downloads and information about the program can be found at the URL \url{xemacs.sourceforge.net}.

Xv

Written, maintained, and copyrighted by John Bradley, xv is an interactive image manipulation program that runs in X-windows. Downloads, information about the program, and information about licensing and registering is available at the URL \url{www.trilon.com}. 
The user of this index should be aware that not only textual discussions but also some of the problems are indexed. Page numbers displayed in Roman type refer to textual discussions; page numbers displayed in Italic type refer to problems.

Symbols

$\text{HEAD}$, 11, 34, 35, 40, 42, 44, 47, 91, 93, 95, 97, 128, 130, 154, 157, 197

$\text{IDLHEAD}$, 11

$\text{NRHEAD}$, 11, 65–67, 93, 95, 97, 130, 133, 157–160

A

aborting execution
in C, 34
accuracy
in evaluating integrals, 123–125, 128–130
in finding roots, 153–154
in solving ODEs, 88–89
with conservation laws, 89
action statements, 15, 18
Adams/Bashforth/Moulton (ABM) methods, 90–91
adaptive methods
for integrals, 125
for ODEs, 90, 93
algorithms
definition, 15
for $a^2 + b^2 = c^2$, 50
for adding, 50
for bowling, 49
for controlling loops, 24
for counting, 24, 25
for exchange, 24
for finding extremes, 25, 26
for integration, 119–128
for Laplace’s equation, 26
for ODEs, 82–91
for parking lot, 48
for root finding, 151–154
for searching, 26, 49
for sorting, 50
for summing, 25
arithmetic operators
in C, 33
array indices
in C, 33
in pseudocode, 17
arrays
for data storage, 28
in C, 33
in pseudocode, 17
scalar, see scalar fields
vector, see vector fields
ASCII character codes, 5
ASCII text files, 8
assigning values to variables
in C, 33
in pseudocode, 18

B

backsubstitution, 162
backward difference, 192
Bessel functions, 138, 139, 169
binary files, 8
binary number system, 4–5
bisection, see root finding
bit, 4–5
black body radiation, 137
blocking statements
in C, 34
in pseudocode, 20, 30
boundary conditions
Dirichlet, 177, 191, 193–194
mixed, 177, 191, 193–194
Neumann, 177, 191
of first kind, see boundary conditions, Dirichlet
of second kind, see boundary conditions, Neumann
of third kind, see boundary conditions, mixed
boundary value problems, 69, 80, 173, 174, 177
byte, 5–6
C

C commands, see also C functions
- `exit`, 228, 238
- `fclose`, 40, 45, 92, 95, 197, 214, 231, 241, 253, 277
- `fgets`, 47
- `fopen`, 35, 40, 45, 47, 92, 95, 96, 197, 214, 231, 241, 253, 277
- `fprintf`, 35, 40, 45, 92, 95–97, 197, 214, 231, 241, 253, 277
- `fscanf`, 47, 48
- `main`, 33, 39, 44
- `return`, 38
- `scanf`, 36, 37, 91, 128, 155, 156, 158, 195, 211, 227, 230, 238, 248, 270

C control structures
- `do/while`, 37, 97
- `for`, 32, 33, 35, 40, 45, 47, 92, 95, 128, 131, 196, 197, 212–214, 228–230, 239, 248, 250, 271, 273, 275–277
- `if/else`, 36, 156, 228, 230, 238, 240, 252, 273, 275
- `switch/case/default`, 36
- `while`, 37, 155, 156

C data types
- `char`, 33, 47
- `FILE`, 35, 39, 44, 47, 197, 214, 231, 241, 253, 277
- `int`, 32, 33, 39, 44, 47, 48, 195, 197, 212, 213, 227–230, 238, 248, 271, 276

C directives, 31
- `define`, 31, 32, 39, 44, 212, 229, 239, 248, 270
- `include`, 31, 32, 39, 44, 195, 196, 211, 227, 228, 238, 248

C functions, see also C commands
- `cos`, 45, 159, 229, 239
- `exp`, 131, 132
- `free`, 196, 228, 238
- `malloc`, 196, 228, 238
- `pow`, 33, 40, 131, 132, 155, 156, 227, 238, 274, 275
- `sin`, 40, 45, 158, 159
- `sizeof`, 228
- `sqrt`, 131, 132
- `user-defined`, 37–38

C header files
- `math.h`, 31, 39, 44, 128, 131, 132, 155, 158, 195, 211, 227, 238, 248
- `nr.h`, 67, 95, 97, 131–133, 157–159
- `nrutil.h`, 95, 97, 157–159

C special symbols
- `!=` (not =), 275
- `\n` (new line), 33, 35, 40, 45, 91, 92, 95–97, 128, 155, 156, 158, 195, 197, 270
- `*` (multiplication), 33
- `+` (addition), 33
- `++` (incrementing index), 48, 128, 131, 196, 197, 228
- `-` (subtraction), 33
- `/` (division), 33
- `/*...*/` (comment), 31
- `==` (logical =), 275
- `\%d` (format), 33, 37, 40, 45, 47, 91, 131, 195, 227, 231, 238, 248
- `\%e` (format), 33, 40, 45
- `\%f` (format), 33, 35–37, 91, 92, 95–97, 131, 132, 155, 156, 158, 195, 197, 227, 229, 231, 238, 248, 270
- `\%g` (format), 48
- `\%s` (format), 131
- `&` (pointer), 36, 37, 91, 97, 155, 156, 158, 160, 195, 227, 238, 270, 276
- `braces`, 34
- `calling FORTRAN from C`, 196–197, 213–214, 276–277
- `cardioid`, 135
- `carrying capacity`, 74
- `case sensitivity`
  - in C, 31
  - in pseudocode, 16
- `cc`, see UNIX commands
INDEX

center of mass
  definition of, 112
  of cardioid, 135
  of semicircular plate, 113
central difference, 192
central processing unit, 3
chaos
  Lorenz system, 104
character codes, 4–5
characteristic equation, 146
classification
  of boundary conditions, 177–178
  of integrals, 111
  of ODEs, 69
  of PDEs, 177
combining statements on one line
  in C, 33
  in pseudocode, 20
commands
  in C, see C commands
  in pseudocode, see pseudocommands
comments
  in C, 31
  in pseudocode, 21, 24
common storage
  in C, see global variables
compiler directives, 17
  in C, see C directives
  in pseudocode, see pseudocode directives
compiling source file, see Local Guide, 16
  in C, 34, 41, 92, 95, 97, 129, 131, 133,
      155–157, 160
cconservation of mass, 186–187
contact information, 315–318
continuing statement on second line
  in C, 33
  in pseudocode, 20
control characters, 4
control structures
  in C, see C control structures
  in pseudocode, see pseudocode control structures
convergence criteria, 28, 88–89, 123–125, 153–154
convergence error, 194–195, 227
Cornu spiral, 116
coupled oscillators, 76–77, 99
Cramer’s rule, 161–162
curve fitting
  to parabola, 171
D
data analysis, 115
data files
  decay_c.dat, 92, 107–108
  decayrk4_c.dat, 95, 108
  decayrkqs_c.dat, 96, 109
  fdm1dla_c.dat, 197, 199–201
  fdmdiffus1d_c.dat, 241–248
  fdmlap2d_c.dat, 253–260
  fdmwave1d_c.dat, 231–237
  fem1dla_c.dat, 214–219
  fem2dla_c.dat, 277, 279–282
  freefall.dat, 172, 173
  irradi_c.dat, 39–41, 47–48, 55–56, 63
  laplace_c.dat, 34–35, 50
  pdens_c.dat, 42, 57–58
  quadpole_c.dat, 47, 61–63
  ruchardt.dat, 174
  wavegd_c.dat, 44, 59–60
data types
  floating point, 6–7
  integer, 6–7
  string, 6–7
data typing
  explicit, 17
  implicit, 17
  in C, 33
  in pseudocode, 16–17, 19–20
default directory, 9
diffraction patterns, 39, 139, 169, 170
diffusion equation
  in 1D, 182–183
  in C, 238–241
  in 2D, 222–224
  in 3D, 183
dimensionless casting
  of integrals, 113–115, 117, 134, 137
  of ODEs, 70–75, 77–79, 145, 147
direct algorithm, 194
discretization
  of Laplace’s equation, 27
  of ODEs, 192–194
  of PDEs, 178, 202
discretization error, 194–195, 227
displaying data on screen
  in C, 32, 33
  in pseudocode, 18
double factorial, 12
dynamic allocation of memory, 17
E
elastic swing, 105
electric field, 116
  in wave guide, 42–44
  of dipole, 53, 54
  of quadrupole, 44–47
electrostatic potential, 116
  of charged disks, 134
  of charged rings, 116–117, 139
  of spherically symmetric distribution, 135
elemental equation in finite element analysis, 207
elliptic integrals, 115, 138
energy levels
  in quantum well, 149, 169
equation of continuity, 186–187
equilibrium points, 144
INDEX

equivalent resistance, 172
error function, 116, 118, 129, 130, 137
Euler’s method, 82–84, 91–92
improved, see improved Euler method
evaluating integrals
with C, 128–133
exchange algorithm, 24
executing program, see Local Guide

F

$f77$, see UNIX commands
factorial, 12
finding roots, see root finding
finite difference approximations
for derivatives, 50
for PDEs, 178
finite difference methods
error in, 194–195
in 1D, 192–202
in 2D, 219–260
with C, 195–202, 227–260
finite element approximations
for PDEs, 178
finite element methods
assembling equations, 203
in 1D, 202–219
in 2D, 260–283
incorporating boundary conditions, 203
with C, 211–219, 269–283
floating point number
double precision, 7
single precision, 6
fluid dynamics, 185
formatting data files, 28–29
formatting output
in C, 33–34, 40, 45
in pseudocode, 18
FORTRAN, 1
forward difference, 192
Fourier series, 119, 136
Fraunhofer diffraction, 38–41
Fresnel integrals, 116, 139
functions
in C, see C functions
typical available, 18

G

Galerkin method, 203, 205–207
gamma function, 12–13
Gauss-Jordan elimination, 203
Gaussian distribution, 115, 137
Gaussian elimination, 162–163, 203
Gaussian integration, see quadrature
gedit, 1
global positioning, 171
global truncation error, see truncation error,
global

global variables
in C, 38, 94
golden ratio, 168
gravitational force, 78

H

harmonic oscillator, 74–75, 98, 100, 103, 198–202
Helmholtz equation, 184
hexadecimal number system, 5
Hooke’s law, 145

IDL, 1
IDL commands
close, 199, 215, 232, 242, 254, 279
findgen, 231, 242, 254, 279
fix, 215, 241, 253
float, 231, 242, 254
fltarr, 215, 231, 241, 253, 279
openr, 199, 215, 231, 241, 253, 279
plot, 199, 215, 232, 242
readf, 199, 215, 231, 241, 253, 279
reverse, 279
string, 232, 242, 254
surface, 254, 279
wait, 232, 242, 254
IDL contact information, 315
IDL control structures
begin/end, 242, 254
while/do, 232, 242, 254
IDL functions
eof, 232, 242, 254
fltarr, 199
reverse, 254
IDL keywords
charsize, 279
format, 232, 242, 254
thick, 199, 215, 232, 242, 254, 279
ticklen, 199, 215
title, 199, 215, 232, 242, 254
xtitle, 279
ytitle, 279
IDL special symbols
not (logical not), 232, 242, 254
IDL system variables
!y.range, 232, 242
improved Euler method, 84–85
inhomogeneities in linear equations, 161
initial value problems, 69, 177
integer
signed, 6
unsigned, 6
interpolation functions, see shape functions
interpolation points, 121, 122
interpreting source code, see also Local Guide, 16
iterative algorithm, 195

K
Kirchoff's laws, 172
Kronecker delta, 118

L

Laguerre's method, see root finding
LAPACK, 1, 270, 276–277
contact information, 315
LAPACK subroutines
agesv.f, 212–214, 276–277
sgtsv.f, 196–197
Laplace transforms, 13–14, 81–82, 100
definition, 13, 81
for ODEs, 81
Laplace's equation, 174, 183, 184
in 2D, 224–227
in C, 248–253
in, 31–35, 50
in pseudocode, 26–28
LATEX, 1
LATEX contact information, 317
least squares fitting, 172–174
lawn mower, 167
Legendre polynomials, 127, 136, 168
Lennard-Jones potential, 171
Lorentz distribution, 134, 138
LSODE, 1
LSODE contact information, 315
LU decomposition, 163, 203

M

machine code, 16
Macintosh OS, 1
MacTeX
contact information, 316
magnetic field, 116
in wave guide, 42–44
of circular loops, 51, 139
magnetic vector potential, 116
MAPLE, 1
MAPLE contact information, 316
MATHEMATICA, 1
MATHEMATICA contact information, 316
mathematical functions
in C, see C functions
MATLAB, 1
MATLAB commands
fclose, 200, 216, 233, 243, 255, 279
feof, 233, 243, 255
fgetl, 254, 255
flipud, 254
fopen, 200, 216, 233, 243, 254, 279
fscanf, 200, 216, 233, 243, 254, 255, 279
fseek, 233, 243
ftell, 233, 243
gca, 233, 243, 279
grid, 200, 216
mesh, 254, 255, 279
meshgrid, 254
pause, 233, 243, 255
plot, 200, 216, 233, 243
rot90, 279
set, 233, 243, 279
title, 200, 216, 233, 243
view, 279
xlabel, 279
ylabel, 279
zlabel, 279
MATLAB contact information, 316
MATLAB control structures
while/end, 233, 243, 255
MATLAB properties
Color, 200, 216, 233, 243
EdgeColor, 254, 255, 279
FontSize, 200, 216, 233, 243, 254, 255, 279
LineWidth, 200, 216, 233, 243, 254, 255, 279
YLim, 233, 243
MATLAB special symbols
! (logical not), 233, 243
% (format), 216
%f (format), 216, 233, 243, 254, 255
(logical not), 255
bof (beginning of file), 233, 243
max-min problems, 144
MAXIMA, 1
MAXIMA contact information, 316
Maxwell-Boltzmann distribution, 137
method of bisection, see root finding
method of least squares, see least squares fitting
midpoint method, 87
midpoint rule, see quadrature
MiKTeX, 1
MiKTeX contact information, 316
ML (left mouse button), 9
MM (middle mouse button), 9
moment of inertia
definition of, 113
of cardioid, 135
of semicircular plate, 113
MR (right mouse button), 9
MUDPACK, 1
contact information, 316
Müller’s method, see root finding
multi-way selection
  in C, 36
  in pseudocode, 21
multistep methods
  for ODEs, 90–91

N

NAG library contact information, 316
named common
  in C, see global variables
natural frequencies
  of spherical cavity, 169
  of stiff bar, 168, 169
  of three-mass system, 167
  of two-mass system, 145–146
Navier-Stokes’ equations, 185
Newton’s method, see root finding
Newton’s second law, 111, 112, 145, 147
  in fluids, 187–189
Newtonian fluid, 189
node numbers
  global, 203
  local, 203
nodes, 177
non-viscous fluid flow, 189
normal distribution, see Gaussian distribution
normal stress
  in fluids, 187–189
numerical approximation
  for first derivative, 192
  for second derivative, 193
numerical integration, see quadrature
Numerical Recipes, 1, 65–68
  for ODEs
    algorithm routines, 93
    driver routines, 93
    stepper routines, 93
  in C, 93–98, 157–160
  broydn.c, 166
  bsstep.c, 93
  caldat.c, 67
  flmoon.c, 66–68
  gaussj.c, 164
  julday.c, 67
  lubksb.c, 164
  ludcmp.c, 164
  mmid.c, 93
  mnewt.c, 166
  newt.c, 166
  nrutil.c, 67, 95, 97, 133, 157, 159
  qromb.c, 133, 140
  qsimp.c, 133, 140
  qtrap.c, 131–133
  rk4.c, 93–95
  rckck.c, 93
  rkqs.c, 93, 96–98
  rtbis.c, 157–160
  rtnewt.c, 157, 159–160
  rtsafe.c, 157, 160
  svbksb.c, 164
  svdcmp.c, 164
  trapzd.c, 130–133
  tridag, 164
  zbrak.c, 160
NUMERICAL RECIPES contact information, 317

nybble, 5

O

object code, 16
octal number system, 5
OCTAVE, 1
OCTAVE commands
  fclose, 200, 217, 235, 246, 257, 281
  feof, 235, 246, 257
  fgetl, 257
  flipud, 257
  fopen, 200, 217, 234, 244, 256, 281
  fscanf, 200, 217, 234, 235, 244, 246, 256, 257, 281
  fseek, 235, 245
  ftell, 234, 244
  gca, 234, 235, 245, 246, 281
  grid, 200, 217
  mesh, 257, 281
  meshgrid, 256
  num2str, 257
  pause, 235, 246, 257
  plot, 200, 217, 234, 235, 245, 246
  rot90, 281
  set, 234, 235, 245, 246, 281
  title, 200, 217, 234, 235, 245, 246, 257
  transpose, 200, 217, 234, 235, 245, 246, 256, 257
  view, 281
  xlabel, 281
  ylabel, 281
  zlabel, 281
OCTAVE contact information, 317
OCTAVE control structures
  while/end, 235, 246, 257
OCTAVE properties
  azim, 281
  color, 200, 217, 234, 235, 245, 246
  edgcolor, 257, 281
  elev, 281
  fontsize, 200, 217, 234, 235, 245, 246, 257, 281
  linewidth, 200, 217, 234, 235, 245, 246, 257, 281
  ylim, 234, 235, 245, 246
OCTAVE special symbols
  ! (logical not), 235, 246, 257
  %d (format), 217, 256, 257
%f (format), 217, 234, 235, 244, 246, 256, 257
bof (beginning of file), 235, 245
ordinary differential equations
with C, 91–98
orthogonal functions, 118–119
OzTeX contact information, 317

P

partial fractions, 99
path (in directory tree), 8
PDEs
classification of, 184–185
pendulum, simple, see simple pendulum
phases of moon, 66–68
pivotal condensation, 163
pivoting, 163
Poisson’s equation, 184
population biology, 74, 105
predictor-corrector methods, 85
procedure, 24
program, 15
projectile motion, see trajectories
proprietary software, 14
pseudocode comments, 21, 24
pseudocode control structures, 15, 20–24
CASE-END_CASE, 21
EXECUTE, 23
FOR–THRU–DO, 22–23, 28–30
FOR–THRU–DO–STEP, 22
IF–THEN–ELSE–END_IF, 3, 4, 21, 25, 26
LOOP–END_LOOP, 3, 4, 21, 24–26
REPEAT–UNTIL, 22
WHILE–DO, 22
pseudocode directives
DIMENSION, 17, 28, 30
pseudocommands
ATTACH, 30
CLOSE, 30
READ, 18
WRITE, 18, 28, 30
PYTHON, 1
PYTHON attributes
append, 201, 218, 281
close, 201, 218, 236–237, 246–248, 281
plot_surface, 258, 282
readline, 201, 218, 236–237, 246–248, 258–260, 281
set_title, 258
set_xlabel, 258, 282
set_ylabel, 258, 282
set_zlabel, 258, 282
split, 201, 218, 236–237, 246–248, 258–260, 281
view_init, 258
PYTHON commands
float, 236–237, 246–248, 258–260, 281
int, 236, 246, 258–260, 281

matplotlib.pyplot
axes, 258–260, 282
clf, 236, 247
figure, 282
grid, 201, 218
pause, 236, 247, 259–260
plot, 201, 218, 236–237, 246–248
title, 201, 218, 236–237, 246–248
ylim, 236–237, 246–248
numpy
array, 201, 218
flip, 258, 282
linspace, 236, 246, 258–260, 282
meshgrid, 258–260, 282
open, 201, 218, 236–237, 246–248, 258–260, 281
range, 258–260, 281
str, 236, 246
PYTHON control structures
for, 201, 218, 258–260, 281
while, 236, 247
PYTHON functions
int, 201, 218
PYTHON keywords
axis, 282
azim, 258
color, 201, 218, 236, 246, 258, 282
decorator, 258, 282
elev, 258
fontsize, 201, 218
linewidth, 201, 218, 236, 246
projection, 258, 282
shade, 258, 282
PYTHON modules
mpl_toolkits.mplot3d, 258, 259, 282
numpy, 201, 218, 236, 246, 258, 259, 282
PYTHON special symbols
!= (not equal), 236, 247
: (range), 201, 218

Q

quadratic formula, 150, 166, 168
quadrature, 119–128
adaptive algorithms, 125
assessing error in, 123–125, 128–130
C programs, 128–133
Gauss-Legendre, 128
Gaussian, 125–128, 137, 139
iterative algorithms, 125
midpoint rule, 120
Newton-Cotes, 119–125
recursive trapezoidal rule, 122
Romberg integration, 123
roundoff error, see roundoff error
Simpson’s rule, 121, 123, 127, 136, 139
trapezoidal rule, 120, 128
truncation error, see truncation error
quantum energy levels, see energy levels
quantum probabilities, 118
  in hydrogen atom, 41–42, 53, 134, 138
  of harmonic oscillator, 118, 137
quantum wave function, see wave function

R
radioactive decay, 73, 83–85, 91–92, 94–98, 102, 104
radius of gyration
  definition of, 113
  of cardioid, 135
railroad problem, 170
range of projectiles, see trajectories
Rayleigh criterion, 170
reaction kinetics, 104
reading data
  from files
    in C, 47–48
    in pseudocode, 31
  from keyboard
    in C, 36–37
    in pseudocode, 18
recasting
  algorithms, 49
  control structures, 48
redirection of output in UNIX, 34
relaxation, 27–28
residual, 154
resonance
  in LRC circuit, 76, 101
Richardson extrapolation, 122
Ritz variational method, 203
Romberg integration, see quadrature
root finding
  accuracy in, 153–154
  by bisection, 151, 156–159
  by Laguerre’s method, 153
  by Müller’s method, 153
  by Newton’s method, 152, 159–160, 165–166, 168, 170
  by secant method, 153
  by trial and error, 154–156, 165
  in C, 157–160, 166
roundoff error, 194–195, 227
  in numerical quadrature, 123
  in ODEs, 88
Ruchardt’s experiment, 174
Runge-Kutta methods, 85–88
running program, see Local Guide

S
scalar fields
  three-dimensional, 41–42
    in C, 41–42
  two-dimensional, 38–41
  in C, 39–41
Schrödinger equation, 80–81, 149, 184
secant method, see root finding
self-adjoint ODE, 191
separating statements on one line
  in C, 33
  in pseudocode, 20
shear stress
  in fluids, 187–189
sigmoid curve, 74
simple pendulum, 103, 114, 138
Simpson’s rule, see quadrature
simultaneous equations
  by direct methods, 163
  by iterative methods, 163
  linear, 161–164
  nonlinear, 164–166
singular equations, 162
singular value decomposition, 164
solving Laplace’s equation, see Laplace’s equation
solving linear equations, 161–164
solving ODEs
  with C, 91–98
sound waves in fluids, 190–191
source code, 16
special symbols
  in C, see C special symbols
speed of sound, 190
spurious roots, 143, 149
stable equilibrium, 144
standing waves
  in cube, 53
  in string, 79–80, 102
Stark effect, 168
structure of data files, 28–29
subroutines
  in C, see C functions

T
Taylor series
  for solving ODEs, 85–89
  for solving simultaneous equations, 166
termination of statements
  in C, 33
  in pseudocode, 20
TeX contact information, 317
TeXlive, 1
TeXlive contact information, 317
TGIF, 1
TGIF contact information, 318
trajectories, 111
double pendulum, 100
earth falling into sun, 134, 137
exponentially decaying force, 134
in central forces, 77–79, 103
in constant magnetic field, 52
in Coulomb field, 101
in crossed $E$ and $B$, 103
in viscous medium, 70–73, 101, 146–147, 167

simple harmonic oscillator, 134
under constant force, 133

trapezoidal rule, see quadrature
tree-structured directories, 8
tridiagonal systems, 163, 173

truncation error
global, 89
in midpoint formula, 124
in numerical quadrature, 123, 131
in ODEs, 88
in Simpson’s rule, 124
in trapezoidal rule, 124, 129
per step, 89

truth table, 19
turning points, 118, 137, 144, 155–157, 169, 170
two’s complement notation, 6
two-way selection
  in C, 36, 156
  in pseudocode, 21

U

UNIX, 1
UNIX commands
  cc, 34, 67, 92, 95, 97, 155–157, 160, 198, 214, 231, 250, 278
  cp, 67
  f77, 241
UNIX special symbols
  . (current directory), 215
  ./ (current directory), 231, 241, 250, 278
  > (output redirection), 34
unstable equilibrium, 144
user-defined functions
  in C, see C functions
user-defined subroutines
  in C, see C functions

V

Van der Pol oscillator, 103
variable names
  in C, 33
  in pseudocode, 16
vector arrays
  three-dimensional, 44–47
  two-dimensional, 42–44
vector fields
  three-dimensional
    in C, 46–47
  two-dimensional
    in C, 44
velocity field
  of rotating object, 54
vertices, 177
vim, 1
viscosity, 189
viscous damping, 71, 74

W

wave equation
  in 1D, 179–181
  in C, 227–231
  in 2D, 219–222
  in 3D, 181–182
wave function, 80–81, 118, 135, 137
weighting functions, 205–206
Windows, 1
Winedt, 1
Winedt contact information, 318
writing data to files
  in pseudocode, 28–31
writing data to screen
  in C, 32, 33
  in pseudocode, 18

X

Xemacs, 1
Xemacs contact information, 318
Xv contact information, 318