COMPUTATION
AND
PROBLEM SOLVING
IN
UNDERGRADUATE PHYSICS
Second Edition

DAVID M. COOK

Department of Physics
Lawrence University
711 E Boldt Way, SPC 24
Appleton, Wisconsin 54911

Copyright © First Edition 2000–16 by David M. Cook
Copyright © Second Edition 2016–20 by David M. Cook

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License (creativecommons.org/licenses/by-nc-sa/4.0/). Any use not permitted by this license requires authorization in writing from David M. Cook.
The first edition of this publication 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 to the Second Edition

Note: Usually with a second edition of a book, the preface to the first edition is preserved and a short preface to the second edition is added. In the present case, the preface to the first edition required here and there a number of edits which, had they not been made, would have been perhaps a bit confusing to readers of the second edition. Consequently, I have elected to depart from the normal practice and simply create a preface for the second edition, though much of its content essentially copies that of the preface to the first edition.

Note: Regardless of which components are included and which omitted in this version of Computation and Problem Solving in Undergraduate Physics, the preface and acknowledgements in the front matter are those from the assemblage containing all components.

Since the mid 1980’s (including the years since my official retirement in 2008), we in the Department of Physics at Lawrence University have been developing and offering curricular components that

- 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

- 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.

Over the years since the mid 1980s, a wide assortment of instructional materials has been drafted and redrafted. This book brings these materials together into a single publication with the hope that it may prove useful to others who seek to achieve these same or similar objectives.

With these objectives in mind, this book consciously focuses on helping students get started. It is not designed to be comprehensive or exhaustive, either in laying out the capabilities of any particular computational resource or in discussing numerical algorithms. Students must understand throughout that they must refer regularly to vendors’ manuals and on-line help files for details beyond those discussed in the book—details that may, in fact, be necessary for successful completion of some of the exercises. The need for that activity is noted here; repeated reminders will not be included in the body of the book.

1The first edition was published in 2003, though it experienced a number of edits and adjustments in subsequent years.
The book is also not a book about computational physics; it addresses uses of computational tools. Indeed, the sophomore course at Lawrence in which students first encounter this book would not in any way replace a course in computational physics. Rather, the materials treated here should provide strong background for a subsequent, junior-senior level course in computational physics, which would—I believe—he substantially enhanced if students came to it already familiar with the resources on which this book concentrates.

One major difficulty in creating materials on computational topics is that different potential users favor different hardware platforms and software packages. Especially in the computational arena, the variety of options and combinations is so great that any single choice (or coordinated set of choices) is bound to limit the usefulness of the product to a small subset of all potential users. This book addresses that difficulty by being assembled from a wide assortment of components, some of which—the generic components—will be included in all versions and others of which—those specific to particular software packages—will be included only if the potential user requests them. Thus, the specific software and hardware discussed in the book can be tailored to the spectrum of resources available at the instructor’s site. Two versions may well differ in numerous respects. One may include the generic components and the components that discuss IDL, MAPLE, C (with Numerical Recipes), and LATEX while another may include the generic components and the components that focus on MATLAB, Mathematica, and FORTRAN (including Numerical Recipes). The table of contents and index contain only entries from the included chapters and sections. To facilitate communication among users of different versions, however, chapter and section numbers and the numbers identifying package-independent exercises are preserved in all versions. In a version that does not include FORTRAN, for example, the FORTRAN sections will be omitted from Chapters 9, 10, 11, 13, 14, and 15 and Chapters 12 and 16—for which FORTRAN is prerequisite—will be omitted altogether. In addition, FORTRAN-specific exercises will be omitted from the end-of-chapter exercises. Because of version-specific omissions such as those just described, there will therefore be gaps in the chapter, section, and exercise numbers in any version that does not include all options. In contrast, within each chapter, equation numbers, figure numbers, table numbers, and footnote numbers advance from one without gaps, and page numbers run continuously from the beginning of the book to the end. In consequence, the numbers assigned to identical equations, figures, tables, footnotes, and pages may differ from version to version, but the numbers assigned to chapters and sections with identical titles and the numbers assigned to identical exercises will be the same in all versions (and will have gaps reflecting omitted chapters, sections, and exercises). Such flexibility would be impossible were we not able to exploit features of LATEX, including the particular capabilities of the \texttt{ifthen} and \texttt{imakeidx} packages, to assemble the PostScript and PDF files that subsequently can be printed to obtain each version.

Even among sites that use the same spectrum of hardware and software, however, some aspects of local environments remain unique to individual sites. Local rules of citizenship; the features and elementary resources of the local operating system; local practices and policies governing structuring of public directories, assignment of accounts and passwords, backup schedules, and after-hours access; licensing restrictions on proprietary software; means to launch particular application programs, compile user-written FORTRAN and/or C programs, and access printers; and numerous other aspects are subject to considerable local variation. This book does not constrain local options in these matters. Instead, its users must draft a site-specific supplement, which we will refer to as the \textit{Local Guide}, to which individuals should refer for site-specific particulars. A LATEX template for that guide, specifically the one used at Lawrence, is available to users of this book, but it will require editing to reflect local practices. In particular, to give local sites flexibility in configuring their environments, we have in the book used symbols like $\texttt{\$HEAD}$, $\texttt{\$IDLHEAD}$, and $\texttt{\$NRHEAD}$ to stand for paths to the specific directories that sit at the head of particular directory trees. All such symbols must be expanded as described in the \textit{Local Guide} when commands or statements illustrated in the book are submitted to the user’s machine.

\footnote{Many of the packages mentioned in this list are commercial and proprietary, and the names are registered trademarks of the respective vendors. Full contact information for all mentioned packages will be found in Appendix Z.}
With the broadest brush, Chapter 1 stands alone and focuses on a number of topics assumed as background for the rest of the book. The next several chapters introduce:

- Specific array processors (Chapter 2 on IDL, Chapter 3 on MATLAB, Chapter 4 on OCTAVE, Chapter 5 on PYTHON),
- Computer algebra systems (Chapter 6 on MAXIMA, Chapter 7 on MAPLE, Chapter 8 on Mathematica),
- Programming languages (Chapter 9—with sections on FORTRAN and C), and
- Subroutine libraries (Chapter 10 on Numerical Recipes, Chapter 12 on LSODE, Chapter 16 on MUDPACK).

The remaining chapters address several important categories of computational processing, specifically:

- Solving ordinary differential equations (Chapters 11 and 12),
- Evaluating integrals (Chapter 13),
- Finding roots (Chapter 14),
- Solving partial differential equations (Chapters 15 and 16)

Each of Chapters 11, 13, 14, and 15 begins with a (generic) section in which several problems drawn from subareas of physics and using the computational technique on which the chapter focuses are laid out. Each of Chapters 11, 13, and 14 then continues with:

- one or more (optional) sections in which some of the identified problems are addressed with whatever computer algebra systems are included in the version,
- a (generic) section on numerical approaches to the category of problem on which the chapter focuses, and
- several (optional) sections in which some of the problems laid out in the first section are addressed with whatever array processors, computer algebra systems, and programming languages are included in the version.

Somewhat in contrast, Chapter 15 continues with:

- a (generic) section on finite difference methods (FDMs) for solving partial differential equations,
- several (optional) sections in which some of the identified problems are addressed using FDMs with each of several tools,
- a (generic) section on finite element methods (FEMs), and
- several (optional) sections in which some of the identified problems are addressed using FEMs.

Every chapter in the book concludes with a collection of exercises using the techniques—both symbolic and numerical—of the chapter. The appendices introduce a publishing system (Appendix A on \LaTeX) and a (UNIX/LINUX) program for producing drawings (Appendix B on TGIF). In time, other options than those available in this edition may be added.

The order of presentation in the book does not compel any particular order of treatment in a course or program of self-study. To be sure, some later sections depend on some earlier sections, but the linkages are not particularly tight. In the Lawrence context, for example, the required sophomore course Computational Mechanics typically covers the chapters and appendices introducing IDL, either MAPLE or Mathematica, and \LaTeX; and finally covers the IDL and either the MAPLE or the Mathematica portions of the chapters on ordinary differential equations (ODEs), integration and root finding. The chapter on Numerical Recipes, the FORTRAN and/or C portions of the chapters

---

4 The second edition has added the shareware programs OCTAVE and PYTHON and replaced the no-longer available commercial program MACSYMA with the shareware program MAXIMA. The addition of OCTAVE and PYTHON explain why Chapters 4–12 in the first edition have become Chapters 6–14 in the second edition.
on programming, ODEs, integration, and root finding, the chapter on partial differential equations and the chapters on LSODE and MUDPACK are the focus of the Lawrence elective junior/senior course *Computational Physics*.

Despite the organization of the chapters by program or by computational technique involved, the focus throughout is on physical contexts. The materials are designed to be used in conjunction with intermediate level courses, not introductory courses. While the illustrations of computational procedures highlight significant physical contexts and most of the examples and suggested exercises emerge from interesting physical situations, the objective is for students to become both fluent and wary in using computational resources in application to these physical situations, not to dwell excessively on the microscopic details of numerical analysis or to teach them the underlying physics (except insofar as successful computer-based solution of problems underscores the power of the fundamental physical ideas). The students are assumed

- to have completed an introductory survey course in physics,
- to have completed courses in calculus, differential equations and, to some extent, linear algebra, and
- to be embarking on intermediate-level studies in physics

as they undertake a study of this book. We focus not so much on the set up of the situations—that is assumed to be the province of other courses—as on computer-based techniques and strategies for determining the solution once the set up is complete. Examples are drawn from classical mechanics, classical electricity and magnetism, thermal physics, quantum mechanics, curve fitting, DC and AC circuit theory, optics, and several other areas.
Acknowledgements

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. Sterne, LU ’97, Mark Nornberg, LU ’98, Scott 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 members 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 Kaufman (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), Kam-Biu Luk (University of California, Berkeley), Daryl Macomb (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
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), Rus-
sel Kaufman (Muhlenberg College), Joelle L. Murray (Linfield College), Shafiq Rahman (Al-
legheny 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 dur-
ing 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 the \LaTeX package \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 \graphicx for including graphic images in \LaTeX documents;
• Leslie Lamport, David Carlisle and other members of the \LaTeX3 team, authors of the \LaTeX package
\ifthen for supporting conditional statements in \LaTeX source files;
• Sebastian Rahtz and Heiko Oberdiek, originator of the \TeX package \hyperref for creating
linked versions of documents as PDF files;
• Paul Vojta, author of \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 \tikz and \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 \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 work-
shop): 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 Uni-
versity), Bob Davis (Taylor University), Mirela Petea (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 Uni-
versity). In July, 2003: Stephen Addison (University of Central Arkansas), Mohan Aggarwal (Alabama A&M), Anne
Caraley (SUNY at Oswego), William Dieterle (University of Pennsylvania), Timothy Duman (University of Indian-
apolis), Brett Fadem (Colby College), Michael Gray (American University), James Meyer (St. Gregory’s Uni-
versity), Prabasaj Paul (Denison University), David Peterson (Francis Marion University), Edward Pogozeiski
(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 Venugopal (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 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), #931248 (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. A small conference held at Lawrence in 1990 and supported by the Alfred P. Sloan Foundation stimulated thinking among the participants about uses of computers in upper-division undergraduate physics. All of these grants have contributed

\footnote{Specifically flmoon, xflmoon, caldat, julday, xjulday, avevar, xavevar, rk4, xrk4, rkqa, rkck, mmid, bstep, rkdoub, odeint, trapzd, xtrapzd, qtrap, xqtrap, qsimp, qromb, point, rtbis, xrtbis, rtmnewt, xrtnewt, rtsafe, xrtsafe, zbrak, gaugej, ludcmp, lubksb, tridag, svdcmp, svbsv, mnewt, newt, and bresyn (both in FORTRAN and in C).}

\footnote{Any 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.}
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 this book. That grant also supported three 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 inserted a footnote in the text, and subsequent printings have incorporated those footnotes.\(^1\) 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 MAPLE details herein apply specifically to MAPLE Version 16 on a Hewlett Packard platform running Windows 7 and a Hewlett-Packard platform running the Fedora 17 implementation of LINUX.
- The IDL details herein apply specifically to 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.
- The Numerical Recipes details in Chapter 10 apply specifically for Version 2.10 and have been tested only on a Hewlett-Packard platform running the Fedora 17 implementation of LINUX.
- The \LaTeX details in Appendix A apply specifically to \LaTeX 2\epsilon with the MiKTeX implementation on a Hewlett-Packard platform running Windows 7. \LaTeX normally responds to the same source code on all platforms.

\(^1\)The date of printing is displayed at the top of the cover page, and I have maintained a dated list of edits made to the source files, so changes made between the original printing and subsequent printings can readily be identified for anyone who wishes to update an outdated printing.
Table of Contents

Preface to the Second Edition iii
Acknowledgements vii
Disclaimer xi
Table of Contents xiii

1 Preliminaries 1
  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

2 Introduction to IDL 15
  2.1 Beginning an IDL Session 15
  2.2 Basic Entities in IDL 16
     2.2.1 Data Types 16
     2.2.2 Variable Names 18
     2.2.3 Assignment of Values to Variables 18
     2.2.4 Commands 19
  2.3 A Sampling of IDL Capabilities 19
     2.3.1 Creating and Examining Arrays 19
     2.3.2 Simple Mathematical Manipulations 23
     2.3.3 A Further Note on Array Indices; Matrix Multiplication 24
     2.3.4 Solving Linear Equations 25
     2.3.5 A First Graph 26
  2.4 Saving and Retrieving the IDL Session 28
  2.5 Loops, Logical Expressions, and Conditionals 31
  2.6 Reading Data from a File 33
  2.7 On-line Help 34
  2.8 Pro-Files 35
  2.9 Eigenvalues and Eigenvectors 39
  2.10 Graphing Scalar Functions of One Variable 40
     2.10.1 The Basic Strategy 41
     2.10.2 Plotting Several Graphs on One Set of Axes 41
     2.10.3 Polar Plots and Custom Axes 42
     2.10.4 Multiple Plots On a Page 44
     2.10.5 Plotting Experimental Data 44

xiii
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.11</td>
<td>Making Hard Copy</td>
<td>47</td>
</tr>
<tr>
<td></td>
<td>. . . of Text</td>
<td>47</td>
</tr>
<tr>
<td></td>
<td>. . . of Graphs</td>
<td>48</td>
</tr>
<tr>
<td>2.12</td>
<td>Graphing Scalar Functions of Two Variables</td>
<td>48</td>
</tr>
<tr>
<td>2.12.1</td>
<td>A Preliminary: The Procedure \texttt{lugen_grid}</td>
<td>49</td>
</tr>
<tr>
<td>2.12.2</td>
<td>Surface Plots: The Procedure \texttt{surface}</td>
<td>50</td>
</tr>
<tr>
<td>2.12.3</td>
<td>Contour Plots: The Procedure \texttt{contour}</td>
<td>52</td>
</tr>
<tr>
<td>2.12.4</td>
<td>Shaded Surfaces: The Procedure \texttt{shade_surf}</td>
<td>53</td>
</tr>
<tr>
<td>2.12.5</td>
<td>The Procedure \texttt{xsurface}</td>
<td>54</td>
</tr>
<tr>
<td>2.12.6</td>
<td>The Procedure \texttt{show3}</td>
<td>55</td>
</tr>
<tr>
<td>2.12.7</td>
<td>Functions of Two Variables in Polar Coordinates</td>
<td>55</td>
</tr>
<tr>
<td>2.13</td>
<td>Graphing Scalar Functions of Three Variables</td>
<td>56</td>
</tr>
<tr>
<td>2.13.1</td>
<td>Reduction to Two-Dimensional Displays</td>
<td>57</td>
</tr>
<tr>
<td>2.13.2</td>
<td>Isosurfaces</td>
<td>60</td>
</tr>
<tr>
<td>2.13.3</td>
<td>Other IDL Routines</td>
<td>61</td>
</tr>
<tr>
<td>2.14</td>
<td>Graphing Vector Fields</td>
<td>62</td>
</tr>
<tr>
<td>2.14.1</td>
<td>The Procedure \texttt{velo___}</td>
<td>63</td>
</tr>
<tr>
<td>2.14.2</td>
<td>The Procedure \texttt{____velo}</td>
<td>63</td>
</tr>
<tr>
<td>2.14.3</td>
<td>The Procedure \texttt{___plot_field}</td>
<td>64</td>
</tr>
<tr>
<td>2.14.4</td>
<td>More Elaborate Displays</td>
<td>65</td>
</tr>
<tr>
<td>2.14.5</td>
<td>Three-Dimensional Vector Fields</td>
<td>65</td>
</tr>
<tr>
<td>2.15</td>
<td>Animation</td>
<td>66</td>
</tr>
<tr>
<td>2.16</td>
<td>Advanced Graphing Features</td>
<td>68</td>
</tr>
<tr>
<td>2.16.1</td>
<td>Fonts</td>
<td>68</td>
</tr>
<tr>
<td>2.16.2</td>
<td>Space Curves</td>
<td>69</td>
</tr>
<tr>
<td>2.16.3</td>
<td>System Variables</td>
<td>71</td>
</tr>
<tr>
<td>2.16.4</td>
<td>Using Multiple Windows</td>
<td>72</td>
</tr>
<tr>
<td>2.16.5</td>
<td>Scaling so Circles are Circles</td>
<td>73</td>
</tr>
<tr>
<td>2.16.6</td>
<td>Working with Color</td>
<td>74</td>
</tr>
<tr>
<td>2.17</td>
<td>Miscellaneous Occasionally Useful Tidbits</td>
<td>76</td>
</tr>
<tr>
<td>2.17.1</td>
<td>Specifying Directories</td>
<td>76</td>
</tr>
<tr>
<td>2.17.2</td>
<td>IDL's Default Directory</td>
<td>76</td>
</tr>
<tr>
<td>2.17.3</td>
<td>Customizing IDL</td>
<td>77</td>
</tr>
<tr>
<td>2.17.4</td>
<td>IDL's Search Path</td>
<td>77</td>
</tr>
<tr>
<td>2.17.5</td>
<td>Restoring IDL's Initial State</td>
<td>78</td>
</tr>
<tr>
<td>2.17.6</td>
<td>Common Storage in IDL</td>
<td>78</td>
</tr>
<tr>
<td>2.17.7</td>
<td>Reading Data from the Keyboard</td>
<td>78</td>
</tr>
<tr>
<td>2.17.8</td>
<td>NaN and Inf</td>
<td>79</td>
</tr>
<tr>
<td>2.18</td>
<td>References</td>
<td>79</td>
</tr>
<tr>
<td>2.19</td>
<td>Exercises</td>
<td>80</td>
</tr>
<tr>
<td>2.19.1</td>
<td>Writing IDL Statements</td>
<td>80</td>
</tr>
<tr>
<td>2.19.2</td>
<td>Finding Eigenvalues and Eigenvectors</td>
<td>81</td>
</tr>
<tr>
<td>2.19.3</td>
<td>Graphing Scalar Functions of a Single Variable</td>
<td>82</td>
</tr>
<tr>
<td>2.19.4</td>
<td>Graphing Scalar Functions of Two Variables</td>
<td>84</td>
</tr>
<tr>
<td>2.19.5</td>
<td>Graphing Scalar Functions of Three Variables</td>
<td>86</td>
</tr>
<tr>
<td>2.19.6</td>
<td>Graphing Vector Fields</td>
<td>86</td>
</tr>
<tr>
<td>2.19.7</td>
<td>Animation</td>
<td>88</td>
</tr>
<tr>
<td>2.20</td>
<td>Listing of \texttt{lugen_grid}</td>
<td>90</td>
</tr>
<tr>
<td>7</td>
<td>Introduction to MAPLE</td>
<td>91</td>
</tr>
<tr>
<td>7.1</td>
<td>Beginning a MAPLE Session</td>
<td>92</td>
</tr>
<tr>
<td>7.2</td>
<td>On-Line Help</td>
<td>94</td>
</tr>
<tr>
<td>7.3</td>
<td>Basic Entities in MAPLE</td>
<td>94</td>
</tr>
<tr>
<td>7.4</td>
<td>Variable Names in MAPLE</td>
<td>95</td>
</tr>
<tr>
<td>7.5</td>
<td>Expressions in MAPLE</td>
<td>95</td>
</tr>
<tr>
<td>7.6</td>
<td>Assigning Values to Variables; Defining Functions</td>
<td>96</td>
</tr>
<tr>
<td>7.7</td>
<td>Creating and Examining Basic Entities</td>
<td>98</td>
</tr>
<tr>
<td>7.7.1</td>
<td>Sequences</td>
<td>98</td>
</tr>
<tr>
<td>7.7.2</td>
<td>Sets</td>
<td>99</td>
</tr>
<tr>
<td>7.7.3</td>
<td>Lists</td>
<td>99</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.7.4</td>
</tr>
<tr>
<td>7.7.5</td>
</tr>
<tr>
<td>7.8</td>
</tr>
<tr>
<td>7.8.1</td>
</tr>
<tr>
<td>7.8.2</td>
</tr>
<tr>
<td>7.8.3</td>
</tr>
<tr>
<td>7.8.4</td>
</tr>
<tr>
<td>7.8.5</td>
</tr>
<tr>
<td>7.8.6</td>
</tr>
<tr>
<td>7.8.7</td>
</tr>
<tr>
<td>7.8.8</td>
</tr>
<tr>
<td>7.8.9</td>
</tr>
<tr>
<td>7.8.10</td>
</tr>
<tr>
<td>7.9</td>
</tr>
<tr>
<td>7.10</td>
</tr>
<tr>
<td>7.11</td>
</tr>
<tr>
<td>7.12</td>
</tr>
<tr>
<td>7.13</td>
</tr>
<tr>
<td>7.13.1</td>
</tr>
<tr>
<td>7.13.2</td>
</tr>
<tr>
<td>7.13.3</td>
</tr>
<tr>
<td>7.13.4</td>
</tr>
<tr>
<td>7.14</td>
</tr>
<tr>
<td>7.15</td>
</tr>
<tr>
<td>7.16</td>
</tr>
<tr>
<td>7.17</td>
</tr>
<tr>
<td>7.17.1</td>
</tr>
<tr>
<td>7.17.2</td>
</tr>
<tr>
<td>7.17.3</td>
</tr>
<tr>
<td>7.17.4</td>
</tr>
<tr>
<td>7.17.5</td>
</tr>
<tr>
<td>7.17.6</td>
</tr>
<tr>
<td>7.18</td>
</tr>
<tr>
<td>7.19</td>
</tr>
<tr>
<td>7.20</td>
</tr>
</tbody>
</table>

9 Introduction to Programming | 157

<table>
<thead>
<tr>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.1</td>
</tr>
<tr>
<td>9.1.1</td>
</tr>
<tr>
<td>9.1.2</td>
</tr>
<tr>
<td>9.1.3</td>
</tr>
<tr>
<td>9.1.4</td>
</tr>
<tr>
<td>9.1.5</td>
</tr>
<tr>
<td>9.2</td>
</tr>
<tr>
<td>9.3</td>
</tr>
<tr>
<td>9.3.1</td>
</tr>
<tr>
<td>9.3.2</td>
</tr>
<tr>
<td>9.4</td>
</tr>
<tr>
<td>9.5</td>
</tr>
<tr>
<td>9.5.1</td>
</tr>
<tr>
<td>9.5.2</td>
</tr>
<tr>
<td>9.5.3</td>
</tr>
<tr>
<td>9.5.4</td>
</tr>
<tr>
<td>9.6</td>
</tr>
<tr>
<td>9.7</td>
</tr>
<tr>
<td>9.7.1</td>
</tr>
<tr>
<td>9.7.2</td>
</tr>
<tr>
<td>9.7.3</td>
</tr>
<tr>
<td>9.7.4</td>
</tr>
<tr>
<td>9.8</td>
</tr>
</tbody>
</table>
### 10 Introduction to Numerical Recipes

10.1 The Numerical Recipes Directory Tree ................................. 237
10.2 Using FORTRAN Recipes ........................................ 238
10.3 Using C Recipes .................................................. 240
10.4 Exercises ......................................................... 241

### 11 Solving Ordinary Differential Equations

11.1 Sample Problems .................................................. 244
  11.1.1 Projectile in a Viscous Medium ................................ 244
  11.1.2 Radioactive Decay ........................................... 247
  11.1.3 Exponential and Logistic Growth .............................. 248
  11.1.4 Forced, Driven, Damped Harmonic Oscillation ............... 248
  11.1.5 An LRC Resonant Circuit .................................. 250
  11.1.6 Coupled Oscillators ....................................... 250
  11.1.7 Motion under Central Forces ................................ 251
  11.1.8 Standing Waves in a String .................................. 253
  11.1.9 The Schrödinger Equation in One Dimension ................ 254

11.2 Laplace Transforms ............................................. 255

11.4 Solving ODEs Symbolically with MAPLE ............................ 256
  11.4.1 Projectile in a Viscous Medium ................................ 257
  11.4.2 Logistic Growth ............................................ 259
  11.4.3 Damped Harmonic Oscillator ................................ 261
  11.4.4 Chain Radioactive Decay .................................... 262
  11.4.5 Coupled Oscillators ........................................ 263
11.6 Algorithms for Solving ODEs Numerically

11.6.1 Euler’s Method ................................................. 267
11.6.2 Improved Euler Method ....................................... 269
11.6.3 Runge-Kutta Methods ......................................... 270
11.6.4 Assessing Accuracy ............................................. 273
11.6.5 Adaptive Methods .............................................. 274
11.6.6 Multistep Methods ............................................. 275

11.7 Solving ODEs Numerically with IDL ................................. 276

11.7.1 Using Elementary Commands .................................. 276
11.7.2 Defining ODEs for IDL .......................................... 279
11.7.3 The Procedures rk4, lsode, ludiffeq_23, and ludiffeq_45 ... 280
11.7.4 Chain Radioactive Decay ..................................... 282
11.7.5 Damped Harmonic Oscillator ................................. 285
11.7.6 Planetary Orbits ................................................ 287
11.7.7 Standing Waves in a String .................................... 290
11.7.8 The Quantum Harmonic Oscillator ........................... 293
11.7.9 Pro-files in the Public Library ................................ 297

11.12 Solving ODEs Numerically with MAPLE .................................. 298

11.14 Solving ODEs Numerically with FORTRAN ......................... 300

11.14.1 FORTRAN Programs from Scratch ......................... 301
11.14.2 Using Numerical Recipes ..................................... 302

11.15 Solving ODEs Numerically with C ................................ 307

11.15.1 C Programs from Scratch ..................................... 307
11.15.2 Using Numerical Recipes ..................................... 309

11.16 Exercises .................................................................. 314

11.16.1 … using Symbolic Methods ..................................... 314
11.16.2 … using Numerical Methods .................................... 318
11.16.3 … using Numerical Recipes ..................................... 321

11.A Listing of decay.f ............................................... 322
11.B Listing of decayrk4.f .............................................. 323
11.C Listing of decayrkqs.f ............................................. 324
11.D Listing of decay.c .................................................. 325
11.E Listing of decayrk4.c ............................................... 326
11.F Listing of decayrkqs.c ............................................. 327

13 Evaluating Integrals ....................................................... 329

13.1 Sample Problems ..................................................... 329

13.1.1 One-Dimensional Trajectories .................................. 329
13.1.2 Center of Mass .................................................... 330
13.1.3 Moment of Inertia; Radius of Gyration ......................... 331
13.1.4 The Large Amplitude Simple Pendulum ....................... 332
13.1.5 Statistical Data Analysis ....................................... 333
13.1.6 The Cornu Spiral ............................................... 334
13.1.7 Electric and Magnetic Fields and Potentials .................. 334
13.1.8 Quantum Probabilities .......................................... 336
13.1.9 Expansion in Orthogonal Functions ............................ 336

13.3 Evaluating Integrals Symbolically with MAPLE ....................... 337

13.3.1 Relativistic Motion Under a Constant Force .................. 338
13.3.2 Center of Mass .................................................... 338
13.3.3 Moment of Inertia; Radius of Gyration ......................... 340
13.3.4 Electrostatic Potential of a Finite Line Charge .................. 340
13.3.5 The Helmholtz Coil .............................................. 342
13.3.6 Period of a Pendulum: Correction as Amplitude Grows .......... 346
13.3.7 Fourier Coefficients for Half-Rectified Signal .................. 347

13.5 Algorithms for Numerical Integration ................................. 348

13.5.1 Newton-Cotes Quadrature ....................................... 349
13.5.2 Rearrangements for Computational Efficiency .................. 351
### TABLE OF CONTENTS

13.5.3 Assessing Error .................................................. 353
13.5.4 Iterative and Adaptive Algorithms .............................. 355
13.5.5 Gaussian Quadrature .............................................. 355
13.6 Evaluating Integrals Numerically with IDL ...................... 358
13.6.1 Using Elementary Commands ................................. 358
13.6.2 The Function luqsimp ........................................... 362
13.6.3 Moment of Inertia .............................................. 363
13.6.4 Quantum Probabilities ......................................... 363
13.6.5 Integrals as Functions of the Upper Limit .................. 364
13.6.6 The Error Function .............................................. 364
13.6.7 The Cornu Spiral ............................................... 365
13.6.8 Integrals as Functions of an Internal Parameter .......... 367
13.6.9 The Off-Axis Electrostatic Potential of Two Rings ....... 367
13.11 Evaluating Integrals Numerically with MAPLE ............... 368
13.11.1 Quantum Probability .......................................... 370
13.11.2 The Error Function ............................................ 370
13.11.3 The Off-Axis Electrostatic Potential of Two Rings ....... 371
13.13 Evaluating Integrals Numerically with FORTRAN .......... 372
13.13.1 Writing Programs from Scratch ............................ 372
13.13.2 Using Numerical Recipes ..................................... 374
13.14 Evaluating Integrals Numerically with C ..................... 377
13.14.1 Writing Programs from Scratch ............................ 378
13.14.2 Using Numerical Recipes ..................................... 379
13.15 Exercises .......................................................... 383
13.15.1 ... using Symbolic Methods ............................... 383
13.15.2 ... using Numerical Methods .............................. 386
13.15.3 ... using Numerical Recipes .................................. 389
13.1 Listing of trapezoidal.f ....................................... 392
13.2 Listing of trapezoidal.c ....................................... 393

14 Finding Roots .......................................................... 395
14.1 Sample Problems .................................................... 395
14.1.1 Classical Turning Points ...................................... 396
14.1.2 A Max-Min Problem: Equilibrium Points .................... 396
14.1.3 Natural Frequencies of Oscillating Systems ............... 397
14.1.4 Range of Projectile in Viscous Medium ..................... 398
14.1.5 Energy Levels in a Quantum Well ........................... 401
14.2 Symbolic Approaches ............................................... 402
14.4 Finding Roots Symbolically with MAPLE ....................... 403
14.4.1 Classical Turning Points ..................................... 404
14.4.2 Equilibrium Points ............................................ 405
14.4.3 Natural Frequencies .......................................... 405
14.4.4 Range of Projectile in Viscous Medium .................... 406
14.6 Algorithms for Finding Roots Numerically ..................... 408
14.6.1 The Method of Bisection .................................... 408
14.6.2 Newton’s Method .............................................. 408
14.6.3 Other Methods .................................................. 409
14.6.4 Assessing Accuracy ............................................ 410
14.7 Finding Roots Numerically with IDL ............................ 411
14.7.1 Classical Turning Points ..................................... 412
14.7.2 Natural Frequencies .......................................... 413
14.7.3 Range of Projectile ............................................ 414
14.7.4 Energy Levels in Quantum Well ............................. 417
14.12 Finding Roots Numerically with MAPLE ....................... 419
14.12.1 Classical Turning Points .................................... 420
14.12.2 Range of Projectile .......................................... 420
14.12.3 Energy Levels in Quantum Well ............................. 422
14.14 Finding Roots Numerically with FORTRAN ................. 424
14.14.1 FORTRAN Programs from Scratch ......................... 424
14.14.2 Using Numerical Recipes .................................... 427
### Table of Contents

A Introduction to \TeX

- A.1 Creating a Simple Document
  - A.1.1 Structuring a \TeX Source File
  - A.1.2 Creating a PostScript File
  - A.1.3 Creating a PDF File
  - A.1.4 Displaying the Document on the Screen
  - A.1.5 Printing the Document

- A.2 Specification of Global Style: The Preamble

- A.3 In-Text Specification of Local Style
  - A.3.1 Type Style
  - A.3.2 Type Size
  - A.3.3 White Space
  - A.3.4 Environments
  - A.3.5 \TeX Packages
  - A.3.6 Miscellaneous Other Capabilities

- A.4 Including Equations
- A.5 Including Lists
- A.6 Including Tables
- A.7 Including Illustrations
  - A.7.1Using Cut and Paste
  - A.7.2Using the graphicx Package
  - A.7.3Using the tikz Package
  - A.7.4Using the picture Environment and pict2e Package

- A.8 Including TOC, LOF, and LOT
- A.9 Including an Index
- A.10 Including Hyperlinks
- A.11 Converting .eps and .ps Files to .pdf
- A.12 Using Conditional Expressions in \TeX
- A.13 Error Messages Generated by \TeX

- A.14 The Page Previewer for .dvi Files
- A.15 The Spell Checker in UNIX
- A.16 A Sample Document
- A.17 Miscellaneous Other Features
- A.18 References
- A.19 Exercises
- A.A Listings
  - A.A.1 ... for Windows
  - A.A.2 ... for UNIX
  - A.A.3 ... for Windows and UNIX

Z Contacting Software Vendors

Index
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 publicly or commercially available subroutines; 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., TpXlive, LpX, MiKTpX, OzTpX, ...) capable of formatting elaborate equations, incorporating PostScript figures, creating 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 students launched into the computational world as sophomores, say, will, as juniors and seniors, ...

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.
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.\(^2\)

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 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, ...—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, ...). 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 all of one step before proceeding to 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 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 Local Guide. This paragraph, however, is the only point in the book at which the wisdom of being logged in is explicitly mentioned.

1.1 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.

\(^2\)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.
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 \(\text{SHIFT}/5\)), 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 \(\textlangle SP\rangle\), has been “printed”.) Pressing the key labeled RETURN “prints” the carriage return character (denoted \(\textlangle CR\rangle\)), 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 \(\textlangle SP\rangle\) 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 \(\textlangle CR\rangle\) 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 \(\textlangle CR\rangle\) will effect the former operation; receipt of a different code, that for the line-feed character \(\textlangle LF\rangle\), 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 \(\textlangle CR\rangle\) from the keyboard must trigger the sending of the codes for the pair of characters \(\textlangle CR\rangle\langle LF\rangle\) to the printer. To simulate a typewriter, we must embellish the above statements to:

3The 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.

4The 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.
PROGRAM TYPEWRITER

LOOP
    Read code from keyboard
    Send code to printer
    IF code is that for <CR>
        THEN Send code for <LF> to printer
    END_IF
END_LOOP
END_PROGRAM

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-off-on-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,

\[
1 \times 128 + 0 \times 64 + 0 \times 32 + 1 \times 16 + 1 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1 = 157
\]

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

6 Actually, the codes will be sent as a stream of bits, each of which is an electrical voltage level that will be either “high” or “lo”, 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) 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 \textit{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 \textit{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.\footnote{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.)}
- a sixteen-bit combination of two consecutive bytes as an \textit{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 \textit{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 \textit{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 \textit{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 \textit{single-precision floating point number}. In the IEEE standard for this format, eight bits (one byte) are assigned to store the proper

\footnote{For purposes of this discussion we will ignore the very considerable differences between (volatile) memory and (non-volatile) hard disks.}
1.1. AN ORIENTATION TO COMPUTERS

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>1</td>
<td>SOH</td>
</tr>
<tr>
<td>2</td>
<td>STX</td>
</tr>
<tr>
<td>3</td>
<td>EOX</td>
</tr>
<tr>
<td>4</td>
<td>EOT</td>
</tr>
<tr>
<td>5</td>
<td>ENQ</td>
</tr>
<tr>
<td>6</td>
<td>ACK</td>
</tr>
<tr>
<td>7</td>
<td>BEL</td>
</tr>
<tr>
<td>8</td>
<td>BS</td>
</tr>
<tr>
<td>9</td>
<td>HT</td>
</tr>
<tr>
<td>10</td>
<td>LF</td>
</tr>
<tr>
<td>11</td>
<td>VT</td>
</tr>
<tr>
<td>12</td>
<td>FF</td>
</tr>
<tr>
<td>13</td>
<td>CR</td>
</tr>
<tr>
<td>14</td>
<td>SO</td>
</tr>
<tr>
<td>15</td>
<td>SI</td>
</tr>
</tbody>
</table>

Table 1.1 (continued):

<table>
<thead>
<tr>
<th>Code</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>DLE</td>
</tr>
<tr>
<td>17</td>
<td>DC1</td>
</tr>
<tr>
<td>18</td>
<td>DC2</td>
</tr>
<tr>
<td>19</td>
<td>DC3</td>
</tr>
<tr>
<td>20</td>
<td>DC4</td>
</tr>
<tr>
<td>21</td>
<td>NAK</td>
</tr>
<tr>
<td>22</td>
<td>SYN</td>
</tr>
<tr>
<td>23</td>
<td>ETB</td>
</tr>
<tr>
<td>24</td>
<td>CAN</td>
</tr>
<tr>
<td>25</td>
<td>EM</td>
</tr>
<tr>
<td>26</td>
<td>SUB</td>
</tr>
<tr>
<td>27</td>
<td>ESC</td>
</tr>
<tr>
<td>28</td>
<td>FS</td>
</tr>
<tr>
<td>29</td>
<td>GS</td>
</tr>
<tr>
<td>30</td>
<td>RS</td>
</tr>
<tr>
<td>31</td>
<td>US</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 \textit{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 \textit{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. \textit{Plus} zero is different from \textit{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. Some files—called ASCII text files—contain nothing but printing ASCII characters (and perhaps such simple control characters as \(\langle\text{CR}\rangle\) and \(\langle\text{HT}\rangle\)) 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, ... 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, ... 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. 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 files 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. 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{} or \LaTeX{} source file; `.f` and `.c` identify files containing source code for FORTRAN and C programs; in this book, `.xc` identifies the executable file generated when a C program is compiled; `.ps` identifies a PostScript file.

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. The Local Guide explains how to translate these symbols if your mouse has fewer than three buttons.\(^\text{13}\) 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

\[
\text{integrate( sin(k*x), x, 0, Pi/2 )}
\]

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 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 SQRTPRO—to return in its second argument the square root of its first argument would be invoked with a statement like

\[
\text{SQRTPRO( } X^2 + Y^2, R \text{ )}
\]

having no variable or equal sign at its beginning.\(^{14}\)

The conventional file type for FORTRAN programs is .f in some operating systems, .for in others. In this book, we use .f.

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. \(x_1\) or \(x_f\), in the statement to be executed but render these variables as subscripted, e.g., \(x_1\) or \(x_f\), in the displayed output.

\(^{14}\)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.
1.5. ASSUMED BACKGROUND

- 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 $\$\text{HEAD}/\text{idl}$, those associated with the chapter on integration will be found in subdirectories of the directory $\$\text{HEAD}/\text{integ}$, 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{::}, \texttt{^=}, \texttt{/\star\star\star/}, and \texttt{!} 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 \( n! \), which is defined when \( n \) is an integer as the product of all integers from \( n \) down to 1, i.e., by
\[
n! = n \times (n - 1) \times (n - 2) \times (n - 3) \times \cdots \times 2 \times 1 \quad (1.1)
\]
is usually familiar. The double factorial function \( n!! \), which is defined (again when \( n \) is an integer) as the product of every other integer, i.e., by
\[
n!! = n \times (n - 2) \times (n - 4) \times (n - 6) \times \cdots \times \begin{cases} 4 \times 2 & n \text{ even} \\ 3 \times 1 & n \text{ odd} \end{cases} \quad (1.2)
\]
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
\[
10!! = 10 \times 8 \times 6 \times 4 \times 2 = 2^5 \times 5 \times 4 \times 3 \times 2 \times 1 = 2^5 \times 5!
\]
\[
\implies (2n)!! = 2^n \times n!
\]
If \( n \) is odd, say 9, recasting its double factorial takes mildly more work, but is illustrated in the chain
\[
9!! = 9 \times 7 \times 5 \times 3 \times 1 = \frac{9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1}{8 \times 6 \times 4 \times 2} = \frac{9!}{8!!} = \frac{9!}{2^4 \times 4!}
\]
\[
\implies (2n + 1)!! = \frac{(2n + 1)!}{2^n \times n!}
\]

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
\[
\Gamma(\nu) = \int_0^\infty t^{\nu-1} e^{-t} \, dt \quad (1.5)
\]
We quickly conclude that
\[
\Gamma(1) = \int_0^\infty e^{-t} \, dt = 1
\]
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
\[
\Gamma\left(\frac{1}{2}\right) = \int_0^\infty \frac{e^{-t}}{\sqrt{t}} \, dt = \int_0^\infty e^{-x^2} \frac{x}{x} \, dx = \int_0^\infty e^{-x^2} \, dx
\]
\[
\Gamma\left(\frac{1}{2}\right) = \int_0^\infty e^{-x^2} \, dx = \int_0^\infty e^{-x^2} \, dx = \int_0^\infty e^{-x^2} \, dx
\]
Then, we examine the square of the quantity of interest, change to polar coordinates, and find that
\[
\left[ \Gamma\left(\frac{1}{2}\right) \right]^2 = \left( \int_0^\infty e^{-x^2} \, dx \right)^2 = \left( \int_0^\infty e^{-x^2} \, dx \right) \left( \int_0^\infty e^{-y^2} \, dy \right)
\]
\[
= \int_0^\infty \int_0^\infty e^{-(x^2+y^2)} \, dx \, dy = \int_0^\infty \int_0^{2\pi} e^{-r^2} r \, dr \, d\phi = \pi
\]
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
\[ \Gamma(\nu) = -\int_0^\infty t^{\nu-1} e^{-t} \, dt = (\nu - 1) \int_0^\infty t^{\nu-2} e^{-t} \, dt = (\nu - 1) \Gamma(\nu - 1) \] (1.11)

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

More generally, a similar argument leads to the conclusion that
\[ \Gamma(n + 1) = n! \] (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.\(^{15}\) 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
\[ \mathcal{L}\left(f(t)\right) = \tilde{f}(s) = \int_0^\infty e^{-st} f(t) \, dt \] (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,
\[ \mathcal{L}\left(af(t) + bg(t)\right) = a \int_0^\infty e^{-st} f(t) \, dt + b \int_0^\infty e^{-st} g(t) \, dt = a \mathcal{L}\left(f(t)\right) + b \mathcal{L}\left(g(t)\right) \] (1.15)

i.e., in more technical terminology, \( \mathcal{L} \) is a linear operator (because integration itself is a linear operation).

\(^{15}\)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 \( \nu > 1 \).
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

$$\frac{d\tilde{f}}{dt}(s) = \int_0^\infty e^{-st} \frac{df(t)}{dt} dt = e^{-sf(0)} + s \int_0^\infty e^{-st} f(t) dt = s\tilde{f}(s) - f(0) \quad (1.16)$$

- 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

$$\frac{d^2\tilde{f}}{dt^2}(s) = s \frac{d\tilde{f}}{dt}(s) - \frac{d\tilde{f}}{dt}(0) = s\left(s\tilde{f}(s) - f(0)\right) - \frac{df}{dt}(0) = s^2\tilde{f}(s) - sf(0) - \frac{df}{dt}(0) \quad (1.17)$$

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 2

Introduction to IDL

Note: All program (*.pro) and data (*.dat) files referred to in this chapter are available in the directory $HEAD/idl, where (as defined in the Local Guide) $HEAD must be replaced by the appropriate path for your site. At some sites, this directory or some other directory containing these files may also have been placed in IDL’s default search path. (See Section 2.17.4.) If so, the files can be found by IDL without explicit specification of a path. Otherwise, you will have to use the full path to copy them into your default directory to access them.

IDL®—an acronym for Interactive Data Language—is an interactive software package whose primary—though certainly not only—capabilities are to manipulate numbers, which may be scalars ($1 \times 1$ arrays), vectors ($1 \times n$ or $n \times 1$ arrays), matrices ($m \times n$ arrays) or higher dimensional arrays, and to produce graphical visualizations. A typical interaction with the program will involve (1) creating one or more arrays in IDL’s workspace (either by computing them directly in IDL or by importing them into IDL from a file created by another program), (2) processing them in some way to produce other arrays, and (3) displaying the end results in an appropriate graphical form. In anticipation of such uses in later chapters, this introduction describes the elementary commands for creating, processing, and displaying arrays. Further details can be found in assorted IDL documentation (see Section 2.18) and the on-line help messages accessible from within IDL itself (see Section 2.7). We shall refer to this documentation collectively as the IDL manuals.

Two user interfaces to IDL are provided, a command-line interface (CLI) and a graphical user interface (GUI), called the IDL Development Environment, which facilitates working with files containing IDL programs. Both interfaces are available in all operating systems, though accessing the CLI in the Windows and Macintosh environments may require special contortions. With either interface, statements to IDL are structured and typed in the same way. Thus, in this chapter, we confine ourselves to describing the CLI.

2.1 Beginning an IDL Session

Detailed instructions for initiating a session with IDL will be found in the Local Guide. In general, IDL will be started either (1) by typing the command `idl` or the command `idlde` at the prompt from the operating system,2 (2) by double-clicking the left mouse button on the IDL icon on the

---

1IDL Exelis Visual Information Solutions, formerly ITT Visual Information Solutions, and formerly Research Systems, Inc., all of Boulder, Colorado, and is registered in the United States Patent and Trademark Office. The acronym IDL also identifies a commercially available program produced and marketed by Exelis. (See Appendix Z for full contact information. More recently, Exelis has become a division of Harris Geospatial Solutions of Melbourne, Florida.) Use of IDL at any particular site is subject to the provisions of whatever license that site has negotiated with Harris or its predecessors. The terms of that license are explained in the Local Guide.

2In this command to the operating system, case may be important.
Presently, the IDL prompt IDL> will appear, either in the window in which the command was typed or in a text entry box at the bottom of the GUI. We must from the outset be aware that

1. Internally, IDL is *not* case sensitive. Except within quoted strings, A and a have the same meaning to IDL.
2. Typing the command exit at the IDL prompt or selecting ‘Exit’ from the File menu in the GUI ends the session.
3. (Control-C) (UNIX; Linux), (Control-Break) (Windows), or (Command-) (Macintosh) instructs IDL to abort its present activity. The activity is not aborted, however, until the statement that was active finishes. Thus, it may take a while before the IDL prompt returns.
4. (Control-D) (UNIX; Linux), (ALT-F4) (Windows), or (Command-Q) (Macintosh) effects an immediate—and computationally rough—exit from IDL to the operating system (UNIX; Linux) or desktop (Windows, Macintosh).
5. Every command line in IDL must end with a (RETURN).
6. Arguments on a line must be separated by commas. If a statement is too long to fit comfortably on one line, the dollar sign ‘$’ at the end of one line can be used to tell IDL that the statement is continued on the next line.
7. The ampersand ‘&’ can be used to separate independent statements on a single line.
8. In any command line, the semicolon ‘;’ can be used to introduce a comment. The semicolon and all following characters in the line will be ignored.
9. An *initial* dollar sign ‘$’ allows temporary escape from IDL to the operating system. Thus, for example, the statement $lp file.ps (UNIX; Linux) or $copy file.ps lpt1: (Windows) issued to IDL will be executed not by IDL but rather by the operating system. Control is then returned to IDL. In the distant past when working on a time-shared terminal, this feature was particularly valuable. In contemporary desktop environments, however, this feature is unnecessary, since we can easily use resources of the desktop to execute other programs without exiting from IDL.
10. Within IDL, we can retrieve previous statements with the up-arrow key and edit individual statements by using the left and right arrow keys to move the cursor in the line, the backspace key to delete the character to the left of the cursor, and other keys to insert characters at the position of the cursor. Further, pressing (RETURN) will execute the entire command line, regardless of where the cursor happens to be positioned within the line.

For illustrative examples of IDL’s capabilities, type demo at the IDL prompt and then surf through the multitude of menus provided. To exit from the IDL Demo Library window, select ‘Quit’ from the File menu in that window.

### 2.2 Basic Entities in IDL

#### 2.2.1 Data Types

While individual items of data can be aggregated into quite complicated data structures, twelve fundamental data types are available in the most recent versions of IDL: 8-bit (unsigned) bytes,
16-bit signed and unsigned integers, 32-bit signed and unsigned integers, 64-bit signed and unsigned integers, 32-bit (single precision) floating-point real numbers, 64-bit (double precision) floating-point real numbers, 32-bit (single precision) floating-point complex numbers, 64-bit (double precision) floating-point complex numbers, and (character) strings. The most important types for our purposes are

- **Bytes**, which occupy 8-bits (1 byte) and store unsigned integer values ranging from 0 to 255. They are also used for logical flags, where odd values (e.g., 1) are treated as ‘true’ and even values (e.g., 0) are treated as ‘false’.

- **Integers**, which occupy 16-bits (2 bytes) and store signed integer values ranging from $-32,768$ through 0 (whose presence explains the asymmetry) to +32,767. Integers can also be used as logical flags, with the same interpretation as described for bytes.

- **Single-precision floating-point (real) numbers**, which have real decimal values with six or seven significant digits and range in absolute value from $10^{-38}$ to $10^{38}$. Each such number requires 32 bits (4 bytes), 24 of which store the (binary form of the) significant digits of the value (including a sign) and 8 of which store the power of two (again including a sign) by which the significant digits must be multiplied to obtain the actual value.

- **Double-precision floating-point (real) numbers**, which have real decimal values with about fourteen significant digits and range in absolute value from $10^{-308}$ to $10^{308}$. Each such number requires 64 bits (8 bytes), 53 of which store the (binary form of the) significant digits of the value (including a sign) and 11 of which store the power of two (again including a sign) by which the significant digits must be multiplied to obtain the actual value.

- **Complex numbers**, which are real-imaginary pairs of either single- or double-precision real floating point values.

- **(Character) strings**, which store text. While each individual character requires 8 bits (1 byte), the variable may store a character string of length ranging from zero up to $2^{31} - 1$ characters.

Although variables can be declared to have a specific data type, we need take no particular action to define the data type of any variable. If no declaration is made, IDL will determine an appropriate data type automatically and dynamically each time a variable is assigned a new value.

We must, however, remain particularly alert to one consequence of dynamic data typing. When using IDL to perform mathematical operations, an explicit decimal point forces floating point representations. Without a decimal point, integer representations will be used. For example, 5/2 effects an integer division (and yields the result 2) whereas 5.0/2.0 effects a floating point division (and yields the result 2.5). Likewise, the expression 200\textsuperscript{2} to square 200 will be evaluated using integer operations and will give an incorrect result ($-25536$) because its actual value exceeds 32767 while 200.0\textsuperscript{2} will be evaluated using floating point operations and will give the correct result (40000.0) stored as a floating point value. To effect floating point operations, we must be certain to include a decimal point or some other cue to prevent IDL from using integer operations by default; failure to include a decimal point may result in unexpected (and incorrect) answers. If an expression uses compatible mixed data types, however, the value of the expression will (usually) be computed after converting all values to the least restrictive data type in the expression and then stored with that data type. For example, the statement \texttt{x = 5.1/2} will be processed by converting the 2 to 2.0, doing a floating division to obtain the result 2.55, and storing the value 2.55 as \texttt{x}. By way of urging caution, however, we note that the statement \texttt{x = 5.1 + 3/2} will return the value 6.1, not 6.6!\footnote{Because division has higher priority than addition, the integer division of 3 by 2 is evaluated before the program realizes that another piece of the expression is clearly a floating point number.}
With one exception, the IDL functions `string`, `byte`, `fix`, `long`, `float`, `double`, `complex`, and `dcomplex` can be used to convert a value of any type to a string, 8-bit unsigned integer, 16-bit signed integer, 32-bit signed integer, single-precision (32-bit) real, double-precision (64-bit) real, single-precision complex, or double-precision complex value, respectively. During some conversions, e.g., the conversion of the floating point value 3.14 to the integer 3 with the expression `fix(3.14)` or the conversion of a complex value to a floating value (which yields simply the real part of the original complex value), information will be lost and cannot be retrieved by invoking the reverse function.

2.2.2 Variable Names

The simplest variables in IDL represent either scalars (single values) or arrays (several values, all of the same data type, referred to by a single name). Variable names can have from one to 255 characters and must start with a letter. The rest of the characters may be letters, numbers, the underscore character, or the dollar sign. Variable names are insensitive to case. Because the data type of a variable is assigned dynamically, a particular variable may have one type at one point in a session and a different type at another point in the same session.

2.2.3 Assignment of Values to Variables

In IDL, the equal sign `=` plays the role of the assignment operator so, for example,

- the statement `alpha = 3.79` creates a single-precision floating-point variable named `alpha` and assigns the value 3.79 to that variable, i.e., stores in a four-byte memory location named `alpha` the single-precision floating-point binary representation of the value 3.79,

- the statement `val = 5` creates a (signed) integer variable named `val` and assigns the value 5 to that variable, i.e., stores in a two-byte memory location named `val` the 16-bit binary representation of the value 5.

- the statement `name = 'David'` and the statement `name = "David"` both create a string variable named `name` and assign the value `David` to that variable, i.e., stores the codes for each of the five characters in five consecutive bytes. Character strings can be enclosed either in single or in double quotation marks.

These assignments only place the indicated value in the identified memory location; they do not automatically display the value on the screen.

Beyond setting variables equal to specific values, a variable can be set equal to an expression with the understanding that IDL will evaluate the expression to determine the value to be assigned to the variable. To achieve that end, of course, all variables used in the construction of an expression must have previously been assigned explicit numerical values. Thus, for example, the statement

\[ x = \alpha + \beta \]

will add the values currently stored in the variables `alpha` and `beta` and store the result in the variable `x`, creating `x` if it does not already exist and overwriting whatever is currently stored in `x`—and perhaps changing its data type—if the variable already exists.

---

5 A string value (e.g., ‘cook’) that does not represent a valid number cannot be converted to a number.
2.2.4 Commands

In IDL the variables identify storage areas in memory and provide ways to refer symbolically to the values stored in those areas. Actual processing of values is effected by one or another of IDL’s commands, the behavior of which is frequently influenced by the value or values of one or more keywords. Some commands—those that invoke what are called IDL procedures—simply accomplish some task (printing an array on the screen, plotting a graph, ...) and do not return a value that must be stored. These commands will be invoked with a statement having the general format

\[
\text{command, argument, argument, keyword=value, keyword=value, ...}
\]

They begin with the command name, which will be followed by one or more arguments, some of which are “free” and others of which are specified by using a named keyword. Other commands—those that invoke what are called IDL functions—return a value determined from their arguments. These commands will be invoked with a statement having the general format

\[
q = \text{command}(\text{argument, argument, keyword=value, keyword=value, ...})
\]

which differs from the first-described format (1) by the assignment of the returned value to a variable—here \(q\)—and (2) by the enclosing of the arguments in parentheses. In both statements, commas are used to separate arguments. When invoking either a function or a procedure, the order of the “free” arguments is mandatory, since the position of each such argument identifies its role. Arguments specified by keywords, however, can be presented in any order, since the keyword identifies the role of the specified value. Further, arguments specified by keywords can be intermingled in any way with “free” arguments. IDL will not be confused.

Most keywords have appropriate default values, so their explicit stipulation is necessary only if the default value is unacceptable. For keywords that control logical (Boolean) values, only the values ‘true’ (1) and ‘false’ (0) are significant. For these keywords, the default is almost always ‘false’, and the shorthand specification ‘/keyword’ rather than the longer specification ‘keyword=1’ can be used to set the keyword to ‘true’.

2.3 A Sampling of IDL Capabilities

In this section, we present several examples illustrating various IDL capabilities and introducing some of the most frequently used commands. The “conversation” in this section should start in a fresh invocation of IDL.

2.3.1 Creating and Examining Arrays

Arrays can be created within IDL or—see Section 2.6—read in from files. Individual elements of arrays can be accessed by specifying the column and row, as in \(\text{array name}[\text{column}, \text{row}]\), where the column and row indices begin at zero (repeat: indices begin at 0), or by specifying the element number, as in \(\text{array name}[\text{element number}]\) where the elements in the array are numbered sequentially starting with zero and then continuing from left to right, top to bottom. The syntax

\[
\text{array name}[\text{column}, \text{row}]
\]

Note that, in IDL, the first index identifies the column and the second index identifies the row in which the element resides. IDL’s convention is the reverse of the more common assignment. Note also that, starting with Version 5.0 of IDL, array indices are enclosed in square brackets to avoid ambiguity. In earlier versions (and still acceptable but not recommended), array indices and arguments of functions were both enclosed in ordinary parentheses, leaving the notation \(\text{data}(3)\), for example, meaning either the fourth entry in an array \(\text{data}\) or the value of the function \(\text{data}\) at argument 3—an ambiguity that the recommended new notation avoids.

We here limit our discussion to arrays of one and two dimensions. See the IDL manuals for ways to create arrays of three or more dimensions.
for creating arrays, assigning values to elements in arrays, and referring to arrays using subscripts is illustrated in the following paragraphs.

The simplest assignments assign single values to variables. We might, for example, create and display simple string and numeric variables with statements like

```
IDL> alpha = 3.79  
Assign floating value to alpha.
IDL> print, alpha  
Display value on screen.
3.79000

IDL> val = 5  
Assign integer value to val.
IDL> print, val  
Display value on screen.
5

IDL> name = ’David’  
Assign string value to name.
IDL> print, name  
Display value on screen.
David
```

Note that we have here introduced the IDL command `print`, which invokes a `procedure`. The command itself is separated from its argument by a comma.

To create vectors and arrays containing more than one numeric element, we must use slightly more complicated statements like

```
IDL> b = [ 1, 0, 3, 7, 10 ]  
Create a (row) vector `b` of `integers` and print it to the screen. To create a floating point vector, at least one value would have to have an explicit decimal point. Entries in the row are separated by commas.
IDL> print, b
1 0 3 7 10

IDL> a = [ [1,2,3], [4,5,6], [7,8,9] ]  
Assign (integer) values to a 3 × 3 matrix and print it to the screen. Values in the inner sets of brackets define the rows. Note the positions of commas and brackets.
IDL> print, a
1 2 3
4 5 6
7 8 9

IDL> data = fltarr(100, 100)  
Create a 100 × 100 array of zeros. Remember, however, that the indices range from 0 to 99. Note also that we have introduced the command `fltarr`, which invokes a `function`. In effect, this statement preallocates space for the array `data`.
```

Many IDL functions that operate on or produce particular data types are complemented by analogous functions that will perform the corresponding operations on the other data types. Thus, complementing the function `fltarr (floating array)`, the functions `bytarr, intarr, lonarr, dblarr, complexarr, dblcomplexarr, and strarr` create arrays of some of the other main data types. Each of these functions creates an array of the specified dimensions full of zeros or—in the case of the string array—of nulls, into which data of the specified type can then be placed.

To remind ourselves of the variables IDL knows at any particular moment and to find out something about those variables we could use the command `help` at the IDL prompt. Given the state created in IDL’s workspace by the commands in this section, we might invoke `help` with no arguments to obtain the report

---

8IDL statements are shown on the left; comments describing the statements are shown on the right. Further, to save space, we will routinely compact IDL’s output by omitting blank lines and extra spaces.

9In contrast to some of its competitors, statements in IDL do not automatically display the results of their execution and no special terminating character is necessary to suppress that output. An explicit `print` statement must be used if explicit output on the screen is desired.

10For simplicity, we have deleted additional information about features we have not yet introduced. Note also that variable names have been displayed entirely in upper case, even though we entered them in lower case.
2.3. A SAMPLING OF IDL CAPABILITIES

IDL> help
A      INT     = Array[3, 3]
ALPHA  FLOAT   = 3.79000
B      INT     = Array[5]
DATA   FLOAT   = Array[100, 100]
NAME   STRING  = 'David'
VAL    INT     = 5

which lists all known variables, their data types, and either their values (for scalars) or their dimensions (for arrays). The statement help, variable_name will display the same information about the specified variable. Many keywords—all of them identified in the IDL manuals—may be used with the help command.

Information about individual variables can also be obtained with the function size, which returns a value that must be assigned to a variable or printed. For example, the statements

IDL> q=1.5 & ln='Cook' & v=intarr(5) & m=fltarr(6,8)
IDL> print, size(q), size(ln), size(v), size(m)

create a floating scalar, a string scalar, an integer vector, and a floating array and then display information about each in order. In all cases, the first value in the (row) vector returned by size is the dimension of the entity, with 0 for scalars, 1 for vectors, and 2 for two-dimensional arrays. Also in all cases, the next to last element codes the data type with, for example, 1 for byte, 2 for integer, 3 for long integer, 4 for single-precision floating point, and 7 for string. Further, the last element in each case is the total number of elements in the variable itself (1 for scalars, 5 for a 5-element vector, 48 for a 6 × 8 array). When present, additional elements (which will be positioned between the first and the penultimate elements) convey the dimensions of the variable (second element 5 for the 5-element vector; second and third elements 6 and 8 for the 6 column × 8 row array).

Should we wish to delete some of the variables to free memory for other purposes, we could use a statement like

IDL> delvar, q, ln, v, m

which removes the variables from IDL’s list and releases the memory their values had occupied.

Once an array has been created, we can access individual elements in the array with statements like

IDL> print, a[0,1]  
  4
IDL> print, a[3]  
  4

Print the value in the first column, second row of a.

IDL> print, a[3] = 10
IDL> print, a  
  1  2  3
  10  5  6
  7  8  9

Assign the value 10 to the fourth element (first column, second row) of a and print the new matrix. The alternative statement a[0,1] = 10 would achieve the same end.
IDL> print, a[*,1]
10 5 6
Print the second row of \texttt{a}. The asterisk is a wild card and stands for all values of the corresponding index.

IDL> c = a[0:1,0:1]
IDL> print, c
1 2
10 5
Extract a $2 \times 2$ matrix made from the four elements in the upper left hand corner of \texttt{a} and print it. The colon between two values tells IDL to use a range of values running in integer steps from the first to the last, inclusive.

Similar to the function \texttt{fltarr}, which returns an array of zeroes, the function \texttt{findgen} (floating index generator) returns a floating point array of the specified dimensions, each element of the array being the element number of that element. For example,

IDL> y = findgen(3)
IDL> print, y
0.00000 1.00000 2.00000
Create a three-element vector with each element set equal to its element number. Note that (1) the values in the array are floating point values and (2) they \textit{start at zero}.

IDL> z = findgen(3,3)
IDL> print, z
0.00000 1.00000 2.00000
3.00000 4.00000 5.00000
6.00000 7.00000 8.00000
Create a $3 \times 3$ matrix (nine elements total), with each element set to its element number. Again, (1) the values in the array are floating point values and (2) they start at zero.

The counterparts of the function \texttt{findgen} for some of the other data types are \texttt{bindgen}, \texttt{cindgen}, \texttt{dcindgen}, \texttt{dindgen}, \texttt{indgen}, \texttt{lindgen}, and \texttt{sindgen}.

Further, once an array has been created, it can be massaged in a variety of ways and various properties of the array can be evaluated. Note, for example the statements

IDL> z = indgen(3,3)
IDL> print, z
0 1 2
3 4 5
6 7 8
Create and display a simple array.

IDL> print, transpose(z)
0 3 6
1 4 7
2 5 8
Evaluate and display the transpose of \texttt{z}.

IDL> print, reverse(z,1)
2 1 0
5 4 3
8 7 6
Evaluate and display \texttt{z} after reversing the order of the columns (second argument = 1).

IDL> print, reverse(z,2)
6 7 8
3 4 5
0 1 2
Evaluate and display \texttt{z} after reversing the order of the rows (second argument = 2).
2.3. A SAMPLING OF IDL CAPABILITIES

IDL> a = [ [4,1,3], [6,2,0], [3,-2, 5] ]
Create a new array.

IDL> print, a
Display array.

4  1  3
6  2  0
3 -2  5

IDL> print, determ(a)
Evaluate and display the determinant of a.
-44.0000

IDL> print, invert(a)
Evaluate and display the inverse of a.

-0.227273  0.250000  0.136364
0.681818 -0.250000 -0.409091
0.409091 -0.250000 -0.0454545

2.3.2 Simple Mathematical Manipulations

The ordinary mathematical operations of combining of variables and constants are achieved with the usual operators: + for addition, - for subtraction, * for multiplication, / for division, and ^ for exponentiation. Within IDL, these operators are given the usual priorities so that, for example, in the expression a*b^2 + c/d, b is first squared, then a is multiplied by b^2 and c is divided by d, and finally c/d is added to ab^2. Parentheses can be used to effect an order of operations different from that dictated by the default priorities.

Further, IDL is aware of the many functions that are important to scientific computation, including the absolute value abs; the square root sqrt; the trigonometric functions sin, cos, and tan; the inverse trigonometric functions asin, acos, and atan; the base-10 and natural logarithms, alog10 and alog; the exponential exp; the hyperbolic functions cosh, sinh, and tanh; and the Bessel functions of integer order beselj, beseli, and besely. Except for the Bessel functions (see the IDL manuals), each is invoked with a statement of the form

q = function_name(x)

after the execution of which the variable q will contain the value of the corresponding function at the argument contained in the variable x. Note particularly that the arguments given to the trigonometric functions must be in radians, and the values returned by the inverse trigonometric functions will be in radians. Note also that the function atan has an alternative two-argument form,

q = atan( y, x )

which will return in q the value atan(y/x) of the corresponding angle in the proper quadrant as determined by the separate signs of x and y.

Continuing from the point reached at the end of Section 2.3.1, the following record of an IDL session demonstrates a few mathematical manipulations:

IDL> eq1 = 8.4*sin(30.0*!pi/180.0)*sqrt(3.0)
Assign the value of an expression to the variable eq1. The variable !pi is a system variable and has the value \pi = 3.14159\ldots. Other system variables are listed in the IDL manuals.

IDL> print, eq1
Print the value of eq1 to the screen. Note that it is a floating point value.
7.27461

IDL> print, byte(eq1)
Convert eq1 to a byte value. Note the loss of information in this conversion.
7

IDL> print, complex(eq1)
Convert eq1 to a complex number. Note that the real and imaginary parts of the answer are separated.
( 7.27461, 0.000000 )
View again the value assigned to \( c \) in Section 2.3.1.

Create a matrix \( d \).

Add \( c \) and \( d \) \textit{element by element}.

Multiply \( c \) and \( d \) \textit{element by element}.

Divide \( c \) and \( d \) \textit{element by element}. Note that \textit{integer} division has been done. To achieve floating division, the elements would be entered with explicit decimal points. Alternatively, at least one array might be “floated” with the function \textit{float}.

### 2.3.3 A Further Note on Array Indices; Matrix Multiplication

Most of the time, we need not be particularly conscious of IDL’s unusual convention in which the \textit{first} index in the symbol \( a[i,j] \) for an element of a matrix is regarded as the \textit{column} index and the \textit{second} index as the \textit{row} index. With matrix multiplication, however, that unusual convention makes a difference. Both for IDL and for the rest of the world, the elements of a matrix product are defined by

\[
e = dc \quad \Rightarrow \quad e_{ij} = \sum_k d_{ik} c_{kj} \quad (2.1)
\]

With this definition, the summation for IDL involves migrating down a column (i.e., from row to row in a fixed column) of the \textit{first} matrix as one migrates across a row (i.e., from column to column in a fixed row) of the \textit{second} matrix. \textit{That} process is specified by the operator \# and, for the matrices \( d \) and \( c \) defined above, yields the result

\[
IDL> e = d \# c
\]
\[
IDL> print, e
\]
\[
17 \quad 12
\]
\[
65 \quad 60
\]

This result is \textit{not} what most people will find in simply writing out the matrices and evaluating the matrix product by hand. \textit{That} operation—migrating from column to column in a fixed row of the first matrix as we migrate from row to row in a fixed column of the second matrix—yields the result

\[
e = dc \quad \Rightarrow \quad \begin{pmatrix} 3 & 4 \\ 7 & 4 \end{pmatrix} \begin{pmatrix} 1 & 2 \\ 10 & 5 \end{pmatrix} = \begin{pmatrix} 3 \times 1 + 4 \times 10 & 3 \times 2 + 4 \times 5 \\ 7 \times 1 + 4 \times 10 & 7 \times 2 + 4 \times 5 \end{pmatrix} = \begin{pmatrix} 43 & 26 \\ 47 & 34 \end{pmatrix} \quad (2.2)
\]

which, in IDL, is effected by the operator \textit{##}, as shown by the statements

\[
IDL> e = d \## c
\]
\[
IDL> print, e
\]
\[
43 \quad 26
\]
\[
47 \quad 34
\]

The second of these results can in fact also be found by transposing the result of IDL-multiplying the transposes of the two factors,
2.3. A SAMPLING OF IDL CAPABILITIES

IDL> print, transpose( transpose(d) # transpose(c) )
43    26
47    34

but the operator ## is distinctly simpler. IDL’s unusual convention can readily cause confusion. Beware!

If (in the usual convention) we (matrix) multiply a row vector into a column vector, we obtain a number that is called the dot product of the n-dimensional vectors. That dot product can be quickly evaluated with the statements

IDL> a = [1,2,3,4,5]  
IDL> b = [6,7,8,9,10]  
IDL> dotprod = a ## transpose(b) 
IDL> print, dotprod
130

Since the two vectors a and b are created as row vectors, the second factor in the dot product must be transposed to convert it into a column vector before the matrix product is evaluated.\footnote{We have used the operator ## for multiplication. We might alternatively have used IDL’s conventions with the statement dotprod = transpose(a) # b.}

2.3.4 Solving Linear Equations

One particular context in which IDL’s row-column switching must be respected emerges when solving simultaneous linear equations. Suppose, for example, we had the equations

\[
\begin{align*}
2x_1 + 5x_2 + 3x_3 &= 3 \\
-x_1 + 3x_2 - 4x_3 &= -4 \\
x_1 - x_2 &= 1
\end{align*}
\]

where the matrix expression uses the usual convention for matrix multiplication. Though the approach ultimately proves to be incorrect, were we not aware of IDL’s convention, we might naively be tempted to enter and verify the coefficient matrix and the vector of inhomogeneities with the statements

IDL> a = [ [2.0,5.0,3.0], [-1.0,3.0,-4.0], [1.0,-1.0,0.0] ]
IDL> print, a
2.00000  5.00000  3.00000
-1.00000  3.00000 -4.00000
 1.00000 -1.00000  0.00000

IDL> b = [3.0, -4.0, 1.0]
IDL> print, b
3.00000 -4.00000  1.00000

Then, recognizing in the abstract that the equation ax = b, where a is a square matrix and x and b are vectors, implies that x = a\(^{-1}\)b, we might—though (from the perspective of numerical accuracy) almost always unwisely—seek a solution with the statements

IDL> ainv = invert(a)
IDL> x = ainv # b
IDL> print, x
-0.0588236 -0.294118  2.82353

We might even—apparently—confirm this solution with the statement

IDL> dotprod = transpose(a) # b.
The solution appears to be correct but, in fact, it is not, as explicit hand substitution into the original equations will reveal. Because of IDL’s internal conventions, we have in fact found and verified the solution to the equations:

\[
\begin{align*}
2x_1 - x_2 + x_3 &= 3 \\
5x_1 + 3x_2 - x_3 &= -4 \\
3x_1 - 4x_2 &= 1
\end{align*}
\] (2.4)

To obtain the correct solution to the original equations, we must express the coefficient matrix with the coefficients in the first equation running down the first column, etc. That is, IDL needs a coefficient matrix that is the transpose of the one we might at first blush imagine. Starting with the matrix a already entered, we would find the correct solution with the statements:

IDL> atran = transpose(a)
IDL> atraninv = invert(atran)
IDL> x = atraninv # b
IDL> print, x
0.852941 -0.147059 0.676471

This solution correctly solves the original equations, as hand substitution will confirm.

Beyond introducing another feature of IDL, this subsection underscores the importance of being careful. *Even things that look right can be wrong for very subtle reasons.*

### 2.3.5 A First Graph

Perhaps the most common use of the function `findgen` is to support the graphing of known univariate functions. First, we need a vector of values spanning the domain of the independent variable. In IDL, the procedure for assigning an array of values equally spaced over some interval to a variable entails (1) deciding the number of segments, $N$, between the minimum and maximum values that are necessary to achieve a smooth plot, (2) calculating the segment size

\[
dx = \frac{(x_{\text{max}} - x_{\text{min}})}{N}
\] (2.5)

and (3) generating the needed vector $x$ containing the values $x_{\text{min}}, x_{\text{min}} + dx, x_{\text{min}} + 2dx, \ldots, x_{\text{min}} + Ndx = x_{\text{max}}$ with the expression

\[
x = x_{\text{min}} + dx \ast \text{findgen}(N + 1)
\] (2.6)

If, for example, we want a graph of the hyperbolic cosine function using 100 divisions over the interval $-3.0 \leq x \leq 3.0$, we would use the IDL statements:

IDL> dx = 6.0 / 100.0

Determine segment size when an interval of length 6 is divided into 100 segments.

IDL> x = -3.0 + dx*findgen(101)

Create a vector of 101 equally spaced values ranging from $-3.0$ to $3.0$. Each element of $x$ differs from adjacent elements by $dx$.

IDL> y = cosh(x)

Assign to each element of a vector $y$ the hyperbolic cosine of the corresponding element of $x$. 

Figure 2.1: Graph of the hyperbolic cosine function.

```
IDL> plot, x, y
```

Plot $y$ versus $x$, drawing straight line segments to connect the points whose coordinates are provided in the two vectors. Note that the vector containing values of the *independent* variable is the *first* argument. Note also that arguments are separated not only from the command but from one another by commas.

The graph created by these statements is shown in Fig. 2.1.

Additional capabilities of the command `plot` are invoked by adding one or more keywords to the command. Four of the most commonly used keywords are

- `charsize` (default 1.0), which controls the size of the characters used in labels and titles;
- `thick` (default 1.0), which controls the thickness of the line drawn for the graph;
- `ticklen` (default 0.02; 1.0 makes marks that span the range of the axes, i.e., draws a full grid), which controls the length of the major tick marks; and
- `title` (default null), which specifies a string value that will title the graph.

Thus we might produce the more readable and useful graph shown in Fig. 2.2 by embellishing the above `plot` command to read

```
IDL> plot, x, y, charsize=1.25, thick=3.0, 
IDL> ticklen=1.0, title='Hyperbolic Cosine Function'
```

Additional capabilities of the `plot` command are discussed in Section 2.10.

Note, incidentally, the power of the statement $y = \cosh(x)$. The argument of the function `cosh` is a *vector* and contains several values. In response to this statement, IDL generates a *second* vector $y$ having the same number of elements as $x$. Each element in $y$ is the hyperbolic cosine of the corresponding element in $x$. In many languages, we would be obliged to use a more elaborate
construction—an explicit loop—to instruct the computer to work on each element individually. IDL automatically understands that intention with this simpler statement. Beyond simpler coding, IDL also achieves faster execution than would be achieved in more traditional coding.

### 2.4 Saving and Retrieving the IDL Session

The command `save` in IDL will save some or all of the variables. For example, the statement

```idl
save, filename='example.dat', a, b, c, /verbose
```

saves the variables `a`, `b`, and `c` in a file named `example.dat` in the default directory.\(^{12,13}\) If no variables are specified, then the command `save` saves all of the variables currently in the IDL workspace; if no filename is specified, the information is saved in the file `idlsave.dat`. The keyword `/verbose`, if present, causes IDL also to save an informative message about each variable.

Saved variables can be retrieved using the `restore` command. To restore the file just saved, for example, one would use the statement\(^{14}\)

```idl
restore, filename='example.dat', /verbose
```

which loads into the IDL workspace all of the variables saved in the file `example.dat` and restores their original names. The optional keyword `/verbose` prints on the screen an informative message about each variable.

The procedure `save` saves IDL variables in a compact, binary, IDL-specific, but machine-independent format. Sometimes we need to save IDL variables as ASCII text files so that other programs such as a text editor can read them. The statements

---

\(^{12}\) The default directory is also referred to as the current directory.

\(^{13}\) See Section 2.17 for a fuller discussion of the default directory and of means to change it.

\(^{14}\) If the default directory is unclear or ill-defined, a specification of the full path to the file may be needed.
write the value(s) in the variable \(a\) in an ASCII text file. Here, the command `openw` opens a file for writing in the default directory, gives it the name `textexample.dat`, and associates with it a Logical Unit Number (LUN), which is used to facilitate subsequent reference to this file.\(^{15}\) The variable \(a\) is then written to the file with the procedure `printf`. Finally, the file associated with LUN 1 is closed. This text file, whose contents is identical to what the statement `print, a` would have displayed on the screen, may then be read by any application that respects the structure IDL gives to the file.

Though one must be careful to have the same default directory, the file created in the previous paragraph can then be read and displayed in a later session with IDL by invoking the statements

```idl
openr, 1, 'textexample.dat'
readf, 1, a
close, 1
print, a
```

One can, of course, include the full path in the quoted string containing the file name and, thereby, avoid concern over the identity of the default directory.

In creating and reading the file `textexample.dat`, we have exploited IDL's capacity to select a format that reflects the character of the data. If we wished to be more explicit about the formatting, we could include the keyword `format` in the `printf` statement. For definiteness, suppose \(a\) stores a \(5 \times 5\) array of integers. Then, we might write that array to a file with the statement

```idl
openw, 1, 'textexample.dat'
printf, 1, a, format = '(5I4)'
close, 1
```

IDL uses a FORTRAN-like style for specifications.\(^{16}\) Here, because the data in \(a\) are integers, the format string uses `I` to indicate integer format and the specification `5I4` conveys that five values are to be written in each line and that each value should occupy four character positions. Since more than five values are to be written, the format specification is reused until all values have been transferred to the output. In the example, the resulting file will have five lines, each containing five values. Additional options in the format string include features to output

- floating point values, e.g., `5f10.4`, which specifies output of five floating point numbers, each occupying a field that is 10 characters wide with four characters after the decimal point.
- varying the format for the numbers, as in

```
printf, 1, a, format = '(I4,I6,I8,I10,I12)'
```

which will output five integers with fields 4, 6, 8, 10, and 12 characters wide, respectively.

\(^{15}\)The LUN must be between one and ninety-nine, since other possible values have predetermined applications, and it cannot be the same as any other user-assigned LUN. Note that the string `textexample.dat` is in quotation marks (single or double) to distinguish it from a variable. Note also that we have explicitly selected a specific LUN for this statement. It would actually be safer for us to have used the statement `openw, unit, 'textexample.dat', /get_lun` so that IDL would assign an available LUN, to which we would refer in subsequent statements with the variable `unit`.

\(^{16}\)Newer versions of IDL also can use C-style format specifications, but the FORTRAN style is IDL's native style.
including text in the output, as in

\[
\text{printf}, 1, \text{a, format } = '("first ", I4, " second", I6,I8,I10,I12)'
\]

In stipulating the field width, remember to allow a character position for a possible minus sign and a character position for a decimal point (if any).

A file produced as described in the previous paragraph can, of course, be examined in any text editor. If the contents of the file are to be imported for processing in some other program, whatever statements import the file must respect its format. For example, to import this \(5 \times 5\) array into a fresh instance of IDL, one would use the statements

\[
\begin{align*}
a &= \text{intarr}(5,5)  \\
\text{openr, 1, 'textexample.dat'}  \\
\text{readf, 1, a}  \\
\text{close, 1}  \\
\text{print, a}
\end{align*}
\]

Define \(a\) with proper dimensions
Open file for reading
Read array from file
Close file
Display array

Note particularly that the variable \(a\) must be defined before it can be referenced in the \text{readf} statement. Note also that white space between values in the file triggers the separation of values when the file is read; the \text{readf} statement need not invoke the keyword \text{format}. Note finally that, had \(a\) in this most recent example been defined as a \(3 \times 3\) array, only the first nine values in the original \(5 \times 5\) array would appear in the created array.

Finally, the command \text{journal} saves to a file everything that is printed in the IDL command window. The one argument to this command names the file and must be a string enclosed in single quotes. For example, to save a session to show the differences among the several matrix products, we might execute the statements\(^{17}\)

\[
\begin{align*}
\text{IDL} & > \text{journal, 'matrix\_journal.pro'}  \\
\text{IDL} & > \text{journal, 'Showing matrix products'}  \\
\text{IDL} & > a = [ [1,3,0], [3,7,2], [0,2,4] ]  \\
\text{IDL} & > \text{print, a}  \\
& \quad 1 \quad 3 \quad 0  \\
& \quad 3 \quad 7 \quad 2  \\
& \quad 0 \quad 2 \quad 4  \\
\text{IDL} & > b = [ [1,2,3], [4,5,6], [7,8,9] ]  \\
\text{IDL} & > \text{print, b}  \\
& \quad 1 \quad 2 \quad 3  \\
& \quad 4 \quad 5 \quad 6  \\
& \quad 7 \quad 8 \quad 9  \\
\text{IDL} & > \text{print, a * b}  \\
& \quad 1 \quad 6 \quad 0  \\
& \quad 12 \quad 35 \quad 12  \\
& \quad 0 \quad 16 \quad 36  \\
\text{IDL} & > \text{print, a # b}  \\
& \quad 7 \quad 23 \quad 16  \\
& \quad 19 \quad 59 \quad 34  \\
& \quad 31 \quad 95 \quad 52  \\
\text{IDL} & > \text{print, a ## b}  \\
& \quad 13 \quad 17 \quad 21  \\
& \quad 45 \quad 57 \quad 69  \\
& \quad 36 \quad 42 \quad 48
\end{align*}
\]

\(^{17}\text{If you are working in a Windows or Macintosh environment and you haven’t yet set the default directory to something outside the IDL directory structure, you probably should make that change before proceeding. See Section 2.17 for information about how to set the default directory.}\)
As illustrated in the second and the last statements, (1) once journaling is in effect the statement journal, ’text’ merely writes the quoted text to the file and (2) the statement journal with no arguments turns journaling off and closes the file.

The file produced by journal can easily be edited in a text editor. As described in Section 2.8, this file can also be executed as a command file to IDL.

### 2.5 Loops, Logical Expressions, and Conditionals

Among the most ubiquitous programming structures is the loop, which provides a means by which a statement or block of statements can be executed some number of times, typically with small changes controlled by a loop index. Several such structures are available in IDL. The simplest is the for loop. Storing the squares of the integers from 1 to 10 in a vector, for example, is readily accomplished with the statements

```
IDL> x = fltarr(10)
IDL> for i = 0, 9 do x[i] = (i+1)^2
IDL> print, x
  1.00000  4.00000  9.00000  16.0000  25.0000
  36.0000  49.0000  64.0000  81.0000 100.0000
```

The first statement establishes the vector, the second steps through all values—integers from 0 through 9, inclusive—of the loop index \(i\), calculating their squares and storing each in the proper element of \(x\), and the third simply displays the end result. With an embellishment of the index, the loop can even be stepped by something other than one, as in

```
IDL> x = fltarr(10)
IDL> for i = 0, 9, 2 do x[i] = (i+1)^2
IDL> print, x
  1.00000  0.00000  9.00000  0.00000  25.0000
  0.00000  49.0000  0.00000  81.0000  0.00000
```

In this version, only elements 0, 2, 4, 6, and 8 are computed; elements 1, 3, 5, 7, and 9 are left at the values established by the function `fltarr`.

If we wanted not only to fill the vector \(x\) but also to see the values as they are computed, we would have to incorporate both the calculation and the print statement in the loop, i.e., we must incorporate two statements in the loop. The proper syntax uses a `begin-end` block and is illustrated in the statements

```
IDL> x = fltarr(10)
IDL> for i = 0, 9, 2 do begin x[i] = (i+1)^2 & print, i, x[i] & endfor
```

In this version, only elements 0, 2, 4, 6, and 8 are computed; elements 1, 3, 5, 7, and 9 are left at the values established by the function `fltarr`.

Note that the vector \(x\) is a row vector. Its display is folded to keep the values on the screen.

```
Note also that the floating point data type of \(x\) established by the first statement is preserved in the loop, even though the expression \((i+1)^2\) will be evaluated with integer operations. Writing \(x[i]=\text{float}(i+1)^2\) would be marginally “cleaner”.
```

```
Alternatively, the statement \(x=findgen(10)\) followed by the statement \(x=(x*1.0)^2\) could replace the first two statements. This alternative would execute slightly faster but would not illustrate the explicit construction of a loop.
```

```
All `begin-end` blocks can be terminated with the simple word `end`. To aid in keeping track of which loop is being terminated at a particular point, however, one can add reminders after the word `end`, as in `endfor` used here.
```
Here, we use the character \& to separate individual statements on a single line and the words begin and endfor to “bracket” the block of instructions to be executed within the loop. Note that there is no ampersand after the word begin but there is an ampersand before the word endfor. Further, for interactive entry of a for loop, IDL must see the entire loop as a single line. For especially long loops, we may find it convenient to present the instructions on several physical lines, using a $-sign at the end of each save the last to tell IDL that the multiline statement constitutes a single logical line. Note, however, that the $-signs will not replace ampersands or commas, which are needed just as if the statement were on a single physical line.

In addition to the for loop, in which we must know ahead of time how many times the loop is to be executed, IDL provides other loops that are capable of deciding when to stop by monitoring the progress of the loop. A while loop to accomplish one of the above tasks might be structured in the form

```
IDL> x = fltarr(10)
IDL> i = 0
IDL> while i lt 10 do begin i=i+1 & x[i-1] = i^2 & endwhile
IDL> print, x
```

```
49.0000 64.0000 81.0000 100.000
```

To construct this while loop, the index i is first initialized to zero. Then, as each pass through the loop begins, that index is tested to see whether it has yet reached the value 10, at which point the loop terminates. Because the index is incremented by one with each pass through the loop, the loop will, in fact, terminate after the tenth pass.

In constructing the while loop in the previous paragraph, we used IDL’s operator lt (less than) to express our first logical condition. IDL, of course, possesses the standard six such operators, specifically lt (less than), gt (greater than), eq (equal to), ne (not equal to), le (less than or equal to), and ge (greater than or equal to). In addition, the operator not toggles logical values between ‘true’ and ‘false’, so, for example, the conditions i lt 10 and not (i ge 10) are equivalent. Finally, Boolean algebra on logical expressions is facilitated by the IDL operators and (logical and), or (logical or), and xor (logical exclusive or).

Logical conditions appear not only in controlling loops but also in structuring branches in a sequence of statements. As with most programming languages, IDL also possesses the if/then/else construct, though the else clause can be omitted if there is in that case nothing to be done. Thus, for example, the statement

```
for i = 0, 9 do if (x[i] lt 0.0) then x[i] = -x[i]
```

will replace each negative element in a vector x with the corresponding positive value, the statement

```
for i = 0, 9 do if (x[i] gt 10.0) then x[i] = 10.0
```

will replace all values greater than 10.0 with the value 10.0, and the statement

```
if a ge 0 then b = a else b = -a
```

---

22When loops are constructed in certain kinds of command files (see Section 2.8), the ampersands and dollar signs are not necessary. Each instruction can then simply be placed on a new line.
will set \( b \) equal to the absolute value of \( a \) (though the function \texttt{abs} will do so more easily). Multiple statements in the \texttt{then} clause will be separated by ampersands and bracketed by \texttt{begin} and \texttt{endif} while multiple statements in the \texttt{else} clause will be separated by ampersands and bracketed by \texttt{begin} and \texttt{endelse}, as in

\begin{verbatim}
if condition then begin statement & statement & endif $
else begin statement & statement & endelse
\end{verbatim}

### 2.6 Reading Data from a File

Frequently, we need to read data into IDL from an appropriately structured ASCII file, which may have been written by another program or created with a text editor. The procedure for reading such files into IDL is described in this section. Before the data can be read, IDL needs information about the type of data and the size and number of arrays in the file. If the file has any headers, IDL needs to be told where to start reading the “real” data.

We illustrate the process with the file\(^\text{23}\) radio.dat, which was created by a FORTRAN program that simulates the decay of a radioactive material into a material that is itself unstable. We must know ahead of time that the file begins with a single line containing labels and continues with 201 additional lines, each containing four numerical values, each separated from the next with one or more spaces.\(^\text{24}\) The first value in each line is a time, and the second, third, and fourth values are the quantities of the initial material, the intermediate material, and the final (stable) material, respectively, at that time.\(^\text{25}\) We read the data from this file with the statements

\begin{verbatim}
IDL> ln = ""                          Define null string for first line.
IDL> quantity = fltarr( 4, 201 )    Create array for data.
IDL> openr, 1, 'radio.dat'           Open the file, assigning logical unit
                                  number (lun)=1.
IDL> readf, 1, ln                     Read first line.
IDL> readf, 1, quantity               Read data, storing it in prepared array.
IDL> close, 1                         Close file.
\end{verbatim}

The first line of the file is now stored in the (string) variable \( \text{ln} \) (and we shall pay no further attention to it), and the numerical values in the file are stored in the 4-column \( \times \) 201-row array \( \text{quantity} \).

We might, for example, plot the time evolution of the quantities of the three materials with the statements

\begin{verbatim}
IDL> plot, quantity[0,*], quantity[1,*], thick=3.0, charsize=1.25, $
IDL> xstyle=8, ystyle=8
IDL> oplot, quantity[0,*], quantity[2,*], thick=3.0
IDL> oplot, quantity[0,*], quantity[3,*], thick=3.0
IDL> xyouts, 2.5, 850.0, 'A'
IDL> xyouts, 2.5, 300.0, 'B'
IDL> xyouts, 40.0, 800.0, 'C'
\end{verbatim}

The first of these statements plots the quantity of the initial material (\( \text{quantity}[1,*] \), the second column of the array) versus time (\( \text{quantity}[0,*] \), the first column of the array), while the second and third statements overlay the quantities of the second and third materials versus time. The remaining three statements, in which locations are expressed in the coordinates of the graph, place the labels \( A \), \( B \), and \( C \) at appropriate points on the graph. The resulting plot is shown in Fig. 2.3.

\(^{23}\) We assume that the file \texttt{radio.dat} is stored in a directory in IDL’s search path (see Section 2.17.4). If that is not the case, then the first statement in the code below must specify the full path to the file, perhaps \$\text{HEAD/idl/radio.dat}. See the note at the beginning of this chapter.

\(^{24}\) Commas or tabs can also be used to separate values in a line.

\(^{25}\) Do not fail to copy this file into your directory and examine its structure with a text editor.
Four previously unmentioned features of IDL have been introduced in this example: (1) The keywords \texttt{xstyle} and \texttt{ystyle} to the \texttt{plot} command provide several possibilities. Here, with the value 8, they simply suppress the “box” style axes in favor of axes drawn on only one side of the graph. Frequently useful other values include 1 (to suppress “nice” scaling on the axis), and 4 (to suppress the axis altogether). A combination of effects results from a sum of values. (2) The command \texttt{oplot}, whose arguments are similar to those of \texttt{plot} (though many keywords available to \texttt{plot} are forbidden to \texttt{oplot}), allows plotting additional graphs without erasing the existing graph. (3) The command \texttt{xyouts}, whose arguments specify a location in the units of the graph and a text string, allows for the placement of text at any point in a graph. (4) A strategy for “reading through” unwanted data in a file in the process of extracting the needed data from the file. In the above example, we recognize the presence of the initial labeling line and read past it.

2.7 On-line Help

IDL provides several ways for users to request help while in an IDL session. As we have already seen, we can obtain information about the current status of the IDL workspace or specific information regarding one or more of the variables stored in that workspace with the command \texttt{help}, perhaps with additional arguments specifying the particular variables about which information is desired, and with the command \texttt{size}. For information about procedures, functions, and keywords in IDL, we can type a question mark (?) at the IDL prompt or select ‘Help Contents’ from the HELP menu in the menu bar of the GUI. IDL responds with a pop-up window with two panels, the left-hand panel providing two tabs (Contents and Index) and the right-hand panel displaying text. Select the appropriate tab by clicking ML on the desired selection and then click ML on the desired entry to bring up the corresponding text in the right-hand panel. If you select the index tab, you can use the search box to locate the desired item. Hot keys to help you narrow your search abound in this help screen. Alternatively, if you know the name of the command on which you seek information, you can type the statement ‘\texttt{?command}’ at the IDL prompt to bring up the help window displaying information on that command directly. However you get there, the information
displayed will describe the syntax of the command—including information on allowed keywords and sample uses. The question mark provides the best route to obtain the most recent documentation on any aspect of IDL.

To display a brief description of routines in the IDL User’s Library (such as the ordinary differential equation or numerical integration routines, or any Lawrence-generated files), type the statement

\[ \text{doc\_library, 'NameOfRoutine'} \]

where \textit{NameOfRoutine} is the name of the routine whose description is desired.

### 2.8 Pro-Files

Any command file called by IDL is saved as a pro-file, so named because the file itself by convention is named with a file type '.pro'. A command file may be a \textit{procedure} file, which defines a procedure controlled by arguments and effecting isolation of all variables internal to the procedure, a \textit{function} file, which defines a function that returns a value to a variable (and also isolates internal variables), or simply a \textit{batch execution} file or \textit{batch} file, which contains a succession of statements that one might alternatively type directly at the command line (and does \textit{not} isolate variables used within the file).

IDL makes available many built-in procedures and functions. We have already introduced several procedures (\texttt{print}, \texttt{printf}, \texttt{plot}, \texttt{openw}, \texttt{openr}, ...) and functions (\texttt{fltarr}, \texttt{invert}, \texttt{findgen}, \texttt{sin}, \texttt{cosh}, \texttt{sqrt}, \texttt{transpose}, ...). Others include procedures for drawing error bars, invoking filters, selecting color tables, and producing animated displays. All are described in the IDL manuals.

IDL’s built-in procedures and functions can be supplemented with user-written batch files, procedures or functions. The batch file is the simplest of these options. Using an available text editor, we merely create a file containing exactly the statements we might otherwise type, one at a time, interactively. For example, we might create a file containing the lines\textsuperscript{26}

\begin{verbatim}
ln = ""  ; Define null string for first line
quantity = fltarr( 4, 201 )  ; Create array for data
openr, 1, 'radio.dat'  ; Open the file, assigning logical unit number (lun)=1
readf, 1, ln  ; Read past first line
readf, 1, quantity  ; Read data, storing it in prepared array.
close, 1  ; Close file
plot, quantity[0, *], quantity[1, *], thick=3.0, charsize=1.25, $  
      xstyle=8, ystyle=8
oplot, quantity[0, *], quantity[2, *], thick=3.0
oplot, quantity[0, *], quantity[3, *], thick=3.0
xyouts, 2.5, 850.0, 'A'
xyouts, 2.5, 300.0, 'B'
xyouts, 40.0, 800.0, 'C'
\end{verbatim}

(Introduced by a semicolon, comments can be included in the file and will be ignored by IDL.) Then, after this file has been stored with the name \texttt{plotradio.pro} (say), either in the default directory or in a directory that is in IDL’s search path (see Section 2.17 and the \textit{Local Guide}), the simple statement\textsuperscript{27}

\texttt{27}

\[ \text{plotradio.pro} \]

---

\textsuperscript{26} We again assume that the file \texttt{radio.dat} resides in a directory in IDL’s search path. For more detail, see footnote 23.

\textsuperscript{27} When no file type is specified, the file type '.pro' is assumed by default.
will instruct IDL to execute all these statements in turn and ultimately produce the graph we have already shown in Fig. 2.3. Especially when we are doing several similar tasks or developing and debugging an extended sequence of statements, the creation of a batch file, which can be easily edited, saved, and then reexecuted, can be an immense time saver.

We can also define customized procedures and functions. The reserved word **PRO** or **FUNCTION** on the first non-blank line of the file informs IDL that a user-defined procedure or function follows. As with batch files, every user-defined procedure or function file should be named with the file type `.pro`. For example, the statements

```plaintext
PRO luplot, funct=tempfunct, N, start, stop

; Procedure luplot is passed a function identified by the
; keyword funct and then, in order, the number of segments into
; which the interval is to be divided, the starting value of
; x, and the stopping value of x for the plot. It returns
; a plot of the function.

; dx = ( stop - start ) / float(N) ; Calculate increment
x = start + dx*findgen(N+1) ; Create vector of independent variable
y = call_function( tempfunct, x ) ; Evaluate dependent variable
plot, x, y, charsize=1.25, thick=3.0, ticklen=1.0 ; Plot graph
RETURN
END
```

define a **procedure** that plots a function passed as a string value `tempfunct` assigned to a user-declared keyword `funct`, with the graph extending over an interval determined by the “free” arguments start and stop and the interval broken into a number of segments specified by the “free” argument N. The procedure is invoked by typing `luplot` followed by the requisite arguments. The procedure will be automatically compiled when it is first called.28 For example, if the file `luplot.pro` is stored in the default directory or in a directory in IDL’s search path, the statement

```plaintext
IDL> luplot, funct='cos', 200, 0.0, 10.0
```

is a legal call to the procedure and would generate a graph of the cosine function \( \cos(x) \) over the interval \( 0 \leq x \leq 10 \), dividing the interval into 200 segments (i.e., plotting 201 points uniformly distributed over the interval).

What if, however, we wanted to use `luplot` to graph a function that is not built into IDL? In that case we would first write a **function** pro-file defining the function of interest. For example, suppose we wanted a graph of the Lorentz lineshape defined by

\[
y(x) = \frac{a^2}{b^2 + (x - x_0)^2}
\] (2.7)

for particular values of \( a \), \( b \), and \( x_0 \). We would begin by writing a **function** pro-file defining the Lorentz lineshape, perhaps using the coding

---

28 Note, however, that we will have to be more explicit about compiling `luplot` if our first execution reveals an error. Our natural instinct would be to fix the error in the file, save the file, and then rerun `luplot`. Unfortunately, at this point, IDL already has knowledge of the (erroneous) version and will not automatically recompile the new version. We will need to invoke the statement `.compile luplot` explicitly to correct IDL’s understanding of the procedure.
2.8. PRO-FILES

Figure 2.4: The Lorentz lineshape.

FUNCTION lineshape, x

;+ 
; The function lineshape returns the value of the Lorentz 
; distribution for given argument x. The parameters a, b, and 
; x0 are explicitly coded in the function.

;a = 1.0 
b = 1.0 
x0 = 2.0 
RETURN, a^2/(b^2 + (x-x0)^2)

END

Once this file has been stored with the name lineshape.pro (say) in the default directory or in
IDL's search path, we simply use a statement like

IDL> luplot, funct='lineshape', 200, -10.0, 10.0

to call luplot once again. In a second or two, the graph of the Lorentz lineshape shown in Fig. 2.4
will appear on the screen.

With the approach of the previous paragraph, we would have to edit the function pro-file and
recompile it if we wanted to graph the Lorentz line shape for a different set of parameters. Because
of the way luplot is written, however, we cannot simply add these parameters as arguments to
lineshape. Were we to adopt that route, we would want to set the parameters a, b, and x0 at IDL's
command level and have them known not within luplot itself but within a routine that luplot
calls. The call statement in luplot does not include the passing of parameters to lineshape. To
achieve this passing of a parameter from IDL's command level into a routine that is not directly
called from the command level, we must make use of named common. In essence, at the command
level, we define a fixed and named area of memory into which we place the desired values of the
parameters at the command level and from which the parameters can be read by any routine that
also identifies the same named area. To this end, we might decide to name the common area `param` (say) and then rewrite `lineshape` to be

```idl
FUNCTION lineshape1, x

;+ ; The function lineshape returns the value of the Lorentz
; distribution for given argument x. The parameters a, b, and
; x0 must be placed in the common area named param prior to invoking
; lineshape1, which then reads the values from that same common
; area.
;-
common param, a, b, x0
RETURN, a^2/(b^2 + (x-x0)^2)
END
```

and store it in the file `lineshape1.pro`. Here, the `common` statement tells the function that it is to find the values of the variables `a`, `b`, and `x0` in the common area named `param`. Then, we invoke `luplot` with this function using the statements

```idl
IDL> common param, a, b, x0
IDL> a = 1.0 & b = 1.0 & x0 = 2.0
IDL> luplot, funct='lineshape1',
IDL> 200, -10.0, 10.0
```

Define common area.
Set parameters.
Graph function

and produce again the graph shown in Fig. 2.4. This time, however, our approach is more flexible, since we can more easily change the values of the parameters and produce another graph.

All of the above pro-files have made use of IDL's capacity for self-documentation. Beyond including comments introduced by semicolons, we have judiciously placed the two lines `;+` and `;->` before and after the descriptive comments at the beginning of each file. Those special lines bracket comments that will be printed when the procedure `doc_library` is invoked with a statement like

```idl
IDL> doc_library, 'luplot'
```

to extract these comments as help messages. *The wisdom of documenting user-written procedures and functions thoroughly cannot be overstressed, and this particular feature of IDL makes it easy to keep the documentation coordinated with—and in the same file as—the coding itself.*

Finally, journal files produced as described in Section 2.4 may also be executed as batch files. When IDL creates a journal file, it automatically puts a semicolon before every line that does not contain a statement. Thus, the file acts as a command file and can be executed using the symbol `@` before the file name.

One distinction between batch pro-files on the one hand and procedure and function pro-files on the other hand must be carefully noted. In Section 2.5, we described the `interactive` statement

```idl
for i = 0, 9, 2 do begin x[i] = (i+1)^2 & print, i, x[i] & endfor
```

which we might have spread over more than one line by writing

```idl
for i = 0, 9, 2 do begin $
  x[i] = (i+1)^2 & $
  print, i, x[i] & $
endfor
```
In a batch pro-file, this statement would have to appear exactly this way, including the dollar signs and ampersands. If this statement were to appear in a procedure or function pro-file, however, it would be written in the form

```idl
for i = 0, 9, 2 do begin
  x[i] = (i+1)^2
  print, i, x[i]
endfor
```

*Neither* the dollar signs nor the ampersands would be included. In function and procedure pro-files, starting a new line is sufficient to start a new statement, though the dollar signs and ampersands can still be used to separate individual statements on a single line and indicate the continuation of a single statement on a second line. In the last form above, the keywords **do** and **begin** must be on the same line (or a dollar sign will be necessary, even in the form for a procedure or function pro-file).

## 2.9 Eigenvalues and Eigenvectors

Eigenvalues and eigenvectors of symmetric matrices figure prominently in many physical contexts (finding quantum mechanical energies and energy shifts, studying small amplitude oscillations, finding principal axes and moments of inertia, ...). The Lawrence-written procedure, **lueigen**, takes as arguments the name of the (symmetric) matrix whose eigenvalues and eigenvectors are desired and the names of the two arrays in which the eigenvalues and eigenvectors are to be stored. Thus, to find eigenvalues and eigenvectors, we need only create the matrix and invoke **lueigen** with statements like:

```idl
IDL> a = [[1,3,0], [3,7,2], [0,2,4]]
Define matrix whose eigenvalues are sought and print it to verify its symmetry.
IDL> print, a
1 3 0
3 7 2
0 2 4
IDL> lueigen, a, evals, evecs
Invoke lueigen.
IDL> print, evals
-0.389315 3.44685 8.94246
IDL> print, evecs
-0.891227 0.310657 0.330465
0.412732 0.253377 0.874902
-0.188062 -0.916129 0.354035
```

Each column of the array **evecs** is one of the *normalized* eigenvectors of the symmetric matrix **a**. To make sure we entered everything properly, we can check the result using the eigenvector equation

\[ a \mathbf{x} = \lambda \mathbf{x} \quad (2.8) \]

where \( \lambda \) is an eigenvalue and \( \mathbf{x} \) is the corresponding eigenvector. For the first eigenvalue and eigenvector, we find that

```idl
IDL> lambda = evals[0]
Extract first eigenvalue.
IDL> x = evecs[0,*]
Extract first eigenvector.
IDL> lhs = a ## x
Evaluate left side of Eq. (2.8).
IDL> rhs = lambda * x
Evaluate right side of Eq. (2.8).
IDL> print, [ lhs, rhs ]
Display left and right sides side by side for comparison.
  0.346968  0.346968
-0.160683 -0.160683
  0.0732154  0.0732154
```

---

\(^{29}\)See the note at the beginning of this chapter.
Table 2.1: The procedure lueigen.pro

```
PRO lueigen, a, eigenvalues, eigenvectors

;+ Procedure lueigen uses the IDL routines trired and triql to find
; the eigenvalues and eigenvectors of a real, symmetric matrix, a.
; Only argument a is supplied as input. The arguments eigenvalues
; and eigenvectors identify the variables into which lueigen places
; its results. Each column of the array eigenvectors is a normalized
; eigenvector of the matrix a. The eigenvector in the ith column
; corresponds to the eigenvalue in the ith element of eigenvalues.
;-

dummy = a ; Protect a
trired, dummy, diagelem, offdiagelem ; Find tridiagonal form
triql, diagelem, offdiagelem, dummy ; Find eigenvectors, values
eigenvalues = diagelem ; Place eigenvalues for output
eigenvectors = transpose(dummy) ; Make eigenvectors columns
RETURN
END
```

Inspection shows that the two columns are the same within differences attributable to internal roundoff.

Behind the scenes, lueigen uses two of IDL’s built-in routines for finding eigenvalues and eigenvectors. First, lueigen invokes trired to reduce the symmetric matrix to an equivalent tridiagonal form which has non-zero elements only in positions on and immediately adjacent to the main diagonal and which, more significantly, has the same eigenvalues and eigenvectors as the original matrix. Then, lueigen invokes triql to compute the eigenvalues and eigenvectors for the transformed matrix. The coding for lueigen, which is implemented as a procedure pro-file, is shown in Table 2.1. In this procedure, trired takes the original matrix—moved to dummy to prevent a from being lost—as input and returns two vectors, diagelem and offdiagelem, containing the diagonal and off-diagonal elements of the equivalent tridiagonal matrix. Then triql takes the diagonal and off-diagonal elements as input and produces the eigenvalues, which are returned in diagelem, and eigenvectors, which are returned in the rows of dummy.

Other IDL routines for dealing with eigenvalues and eigenvectors include eigenql (for real, symmetric arrays), eigenvec (for real, non-symmetric arrays), elmhes (to reduce a real, non-symmetric array to a special form known as the upper-Hessenberg form) and hqr (for finding eigenvalues—but not eigenvectors—of matrices in upper-Hessenberg form). Details about these routines can be found in the IDL manuals.

### 2.10 Graphing Scalar Functions of One Variable

The most common—and simplest—graph is a two-dimensional plot of a function of one variable, i.e., a graph of dependent variable versus independent variable. The basic command for producing such a graph is `plot`, which—in its simplest form—has the syntax

```
plot, independent variable, dependent variable
```

where the two arguments identify vectors in which values of the variables to be plotted are stored. The graph itself is constructed by drawing straight line segments connecting consecutive points
Figure 2.5: On-axis magnetic field of a circular current loop.

\[ B(z) = \frac{1}{(1 + z^2)^{3/2}} \]  

2.10.1 The Basic Strategy

As a first example, suppose that we desire a graph of the magnetic field on the z axis for a circular current loop in the \( xy \) plane. As a function of position, this field is given in dimensionless form by

\[ B(z) = \frac{1}{(1 + z^2)^{3/2}} \]  

Suppose that we want a graph over the interval \(-4.0 \leq z \leq 4.0\) with 100 divisions of the interval (101 points plotted). We would then execute the IDL statements

IDL> dz = 8.0/100.0  
IDL> z = -4.0 + dz * findgen(101)  
IDL> B = (1.0 + z^2)^(-1.5)  
IDL> plot, z, B, title='Magnetic Field on z axis',  
   &xtitle='Dimensionless Position', ytitle='Magnetic Field',  
   &thick=3.0, charsize=1.25, ticklen=1.0

where the keywords title, thick, charsize, and ticklen have the effects described in Section 2.3.5 and the newly introduced keywords xtitle and ytitle label the \( x \) and \( y \) axes, respectively. This sequence of statements will produce the graph shown in Fig. 2.5.

2.10.2 Plotting Several Graphs on One Set of Axes

IDL’s procedure oplot provides the capacity for plotting more than one graph on a single set of axes. When overplotting, a different line style may be used to draw each of the graphs. The plot of undamped and damped sine waves on the same axes shown in Fig. 2.6 is produced with the statements
Table 2.2: Available line styles for use with the keyword linestyle.

<table>
<thead>
<tr>
<th>Value for linestyle</th>
<th>Line Style Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Solid</td>
</tr>
<tr>
<td>1</td>
<td>Dotted</td>
</tr>
<tr>
<td>2</td>
<td>Dashed</td>
</tr>
<tr>
<td>3</td>
<td>Dash Dot</td>
</tr>
<tr>
<td>4</td>
<td>Dash Dot Dot Dot Dot</td>
</tr>
<tr>
<td>5</td>
<td>Long Dashes</td>
</tr>
</tbody>
</table>

Figure 2.6: Damped and Undamped sine waves.

IDL> dx = 20.0/100.0
IDL> x = dx*findgen(101)
IDL> sine = sin(x)
IDL> dampsine = exp(-x/10.0) * sin(x)
IDL> plot, x, sine, title='Damped and Undamped Sine Waves', $ 
IDL> thick=3.0, charsize=1.25
IDL> oplot, x, dampsine, linestyle=4, thick=3.0

The keyword linestyle, which can be used both in the command plot and in the command oplot, specifies a line style other than the default (a solid line). Table 2.2 enumerates the available line styles.

2.10.3 Polar Plots and Custom Axes

IDL is also capable of producing polar plots and of using different axis styles instead of the standard box style. Polar plots are accomplished by specifying the radial ($r$) and polar ($\theta$) coordinates in the execution of the plot command along with the keyword /polar.\textsuperscript{30} For example, to graph the

\textsuperscript{30}The keyword polar is a Boolean (logical) variable, and the specification /polar is equivalent to the specification polar = 1; either sets the keyword to ‘true’. 
cardioid defined in polar coordinates by the equation
\[ r(\theta) = a(1 - \cos \theta) \]  
(2.10)

and shown in Fig. 2.7, we might execute the statements

IDL> dt = 2.0*!pi/100.0  
IDL> theta = dt*findgen(101)  
IDL> a = 3.0  
IDL> r = a*(1.0 - cos(theta))  
IDL> plot, r, theta, /polar, $  
IDL> xstyle=4, ystyle=4, $  
IDL> subtitle='The Cardioid', $  
IDL> thick=3.0, charsize=1.25  
IDL> axis, 0.0, 0.0, xaxis=0  
IDL> axis, 0.0, 0.0, yaxis=0

Here, in the command plot, the radial coordinate comes first and the azimuthal coordinate comes second, the keywords xstyle and ystyle have been set to the value 4 to suppress axes that would otherwise be drawn (inappropriately) as a box around the entire figure, and we have used the keyword subtitle, which differs from the keyword title because it uses a slightly smaller font and it places the title at the bottom of the graph. Having suppressed the automatically drawn axes in the command plot, we then invoke the command axis to draw a new x axis, centered at the origin, and a new y axis, also centered at the origin. In the first invocation of axis, the keyword xaxis is set to indicate that an x axis is to be drawn and set specifically to zero so that the labels are placed below the axis and the ticks are drawn upwards from the axis. In the second invocation, the keyword yaxis is set to indicate that a y axis is to be drawn and set specifically to zero so that the labels are placed to the left of the axis and the ticks are drawn to the right from the axis.\footnote{Setting xaxis to 1 places the labels above and draws the tick marks down from the x axis; setting yaxis to 1 places the labels to the right of and draws the tick marks to the left from the y axis.}

The graph of Fig. 2.7 is, however, distorted because, despite the equal ranges on the two axes, equal coordinate increments in the two directions do not correspond to equal lengths on the two axes. The graph is stretched horizontally by the aspect ratio of the display. Adjusting this aspect ratio is discussed at some length in Section 2.16.5, where several additional keywords available in some of the graphics commands, including the plot command, are introduced.
2.10.4 Multiple Plots On a Page

Sometimes we wish to plot several graphs on one page. IDL gives us substantial flexibility in formatting the layout of these graphs by providing the system variable \texttt{!p.multi}, which controls how many plots appear, how they are formatted, and in what order they are drawn. \texttt{!p.multi} is a five-element integer vector. For our purposes, the first three elements are the important ones. These are

- \texttt{!p.multi[0]}, which keeps track of which segment of the graph window will be used next,
- \texttt{!p.multi[1]}, which specifies the number of plot columns per page,\textsuperscript{32} and
- \texttt{!p.multi[2]}, which specifies the number of plot rows per page.\textsuperscript{33}

If more than two plots are placed in either the horizontal or vertical directions, then the character size used to annotate the graph is decreased by fifty percent.

As a quick example using \texttt{!p.multi}, suppose we wish to plot the sine and cosine functions along with their analogous hyperbolic functions. If we set \texttt{!p.multi[0]=0, !p.multi[1]=2} and \texttt{!p.multi[2]=2}, we will be able to plot two columns and two rows of graphs on one page. The first plot is drawn in the upper left-hand corner of the plot window, and the following plots are placed left to right, top to bottom. The plot shown in Fig. 2.8 is produced with the statements

\begin{verbatim}
IDL> dz = 8.0/100.0
IDL> z = -4.0 + dz*findgen(101)
IDL> v = sin(z) & w = cos(z) & x=sinh(z) & y=cosh(z)
IDL> plot, z, v, title='Sine', thick=3.0
IDL> plot, z, w, title='Cosine', thick=3.0
IDL> plot, z, x, title='Hyperbolic Sine', thick=3.0
IDL> plot, z, y, title='Hyperbolic Cosine', thick=3.0
\end{verbatim}

Note (1) that the ampersand ‘\&’ separates individual statements placed on a single line, (2) that \texttt{!p.multi} is reset to specify one plot per page at the end of this sequence of statements, and (3) that the variable \texttt{!p.multi[0]} is adjusted automatically as the graphing proceeds but can be reset to zero at any time so that the next invocation of \texttt{plot} will erase the current display and restart the graphing at the upper left segment of the graph window.

2.10.5 Plotting Experimental Data

As a final example of two-dimensional graphing, we describe IDL’s ability to produce plots of experimental data complete with error bars, representing each of the data points with a plotting symbol not connected with lines and using logarithmic scales on either (or both) axes. IDL will also allow us to set the range of the axes and annotate the graph with a legend. This example uses data from an experiment on an RC high-pass filter. Suppose the file \texttt{rcdata.dat} containing the data was created with a text editor and stored in the default directory.\textsuperscript{34} When printed, the file produces a two-column display, the first column containing the frequencies at which gains were measured and the second containing the measured gains. In all, the file contains twenty-one measurements (twenty-one lines).\textsuperscript{35} The data are read into the IDL workspace with the statements

\begin{verbatim}
32\text{If } \texttt{!p.multi[1]} \leq 0, \text{ one column of plots is assumed.}
33\text{If } \texttt{!p.multi[2]} \leq 0 \text{ one row of plots is assumed.}
34\text{See footnote 23.}
35\text{Do not fail to copy this file from $\textit{HEAD/idl}$ into your directory and examine its structure with a text editor.}
\end{verbatim}
Figure 2.8: Graphs of sine, cosine, hyperbolic sine, and hyperbolic cosine on the interval $-4 \leq x \leq 4$.

IDL> openr, 1, 'rcdata.dat'  
Open data file.

IDL> data = fltarr(2,21)  
Create variable to receive data.

IDL> readf, 1, data  
Read data.

IDL> close, 1  
Close file.

Next, we plot the data using the statement

```
IDL> plot, /xlog, data[0,*], data[1,*], psym = 4, charsize=1.25,  
IDL> yrange=[0.0, 1.1], ystyle=1,  
IDL> title='Experimental Data for High Pass Filter',  
IDL> xtitle='Frequency', ytitle='Gain (Vout/Vin)'
```

This statement plots the values in the second column of the array data (gain measurements, the dependent variable) versus the values in the first column (frequency measurements, the independent variable) and specifies with the keyword xlog that a logarithmic scale be used on the x axis but preserves a linear scale on the y axis. Using the coding in Table 2.3, the keyword psym specifies which of several symbols is to be placed at each data point. We have also used the keyword yrange, which is set to a two-element vector, to wrest control of the range plotted on the y axis from IDL by specifying the minimum (first element) and maximum (second element) of the desired range. Further, we have set the (already introduced) keyword ystyle to 1 to force IDL not to round the specified limits to “nice” values. The full graph, whose beginnings have been created by the above statements (and whose completion will be described in what follows), is presented in Fig. 2.9.

A graph of experimental data, however, is not complete without error bars. IDL supplies a routine called errplot to overplot error bars on a graph. The statements used to add the error bars are

```
IDL> unc = data[1,*]*0.05  
Compute uncertainties at 5%.

IDL> low = data[1,*] - unc  
Compute lower values.

IDL> high = data[1,*] + unc  
Compute upper values.

IDL> errplot, data[0,*], low, high, width=0.02  
Draw error bars.
```

The keyword ylog can be set to true to specify a logarithmic scale on the y axis. The older commands plot.ol for log x and linear y, plot.io for linear x and log y, and plot.o0 for log x and log y axes have been superseded with the keywords xlog and ylog.

For a more complete description of the errplot procedure, see the IDL manuals.
Table 2.3: Available symbols for use with the keyword `psym`.

<table>
<thead>
<tr>
<th>Value of <code>psym</code></th>
<th>Symbol Drawn</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Plus sign</td>
</tr>
<tr>
<td>2</td>
<td>Asterisk</td>
</tr>
<tr>
<td>3</td>
<td>Period</td>
</tr>
<tr>
<td>4</td>
<td>Open diamond</td>
</tr>
<tr>
<td>5</td>
<td>Open triangle</td>
</tr>
<tr>
<td>6</td>
<td>Open square</td>
</tr>
<tr>
<td>7</td>
<td>Times sign</td>
</tr>
<tr>
<td>8</td>
<td>User-defined symbol; see <code>usersym</code> procedure in the IDL manuals</td>
</tr>
</tbody>
</table>

Procedure `errplot` requires as input three arguments, the first a vector containing the horizontal—yes, horizontal—coordinates of the plotted points, the second a vector containing the lower bound of the value at each data point, and the third a vector containing the upper bound of the value at each data point. Assuming a 5% uncertainty, we calculate first the uncertainty and then the upper and lower values for each data point. (These vectors might alternatively be read from a properly expanded input file.) Then, we invoke `errplot`. A third optional keyword, `width`, is included to specify the length of the horizontal bars drawn at the top and bottom of the vertical error bars as a fraction of the width of the plot window. (The default value is 0.01.)

A graph of experimental data should also include a plot of predicted theoretical results. The equation for the theoretical gain of this circuit is

$$ G = \frac{V_{out}}{V_{in}} = \frac{1}{\sqrt{1 + \left(\frac{f_0}{f}\right)^2}} $$

where $f_0$ is defined as

$$ f_0 = \frac{1}{2\pi RC} $$

(2.11)

Because we are using a logarithmic scale on the horizontal axis, however, we must calculate the vector of independent variables carefully. The variable $f$ needs to have a range from $10^{1}$ Hz to $10^{4}$ Hz. For the most satisfactory plotting, we need to create a vector of values that will represent this range uniformly on a logarithmic scale. To achieve that objective, we begin by creating a vector—call it `temp`—containing a suitable number of values uniformly distributed on a linear scale from 1.0 to 4.0. Then, we take the values of $f$ to be the base-10 exponential of the values in `temp`, i.e.

$$ f = 10^{\text{temp}} = \left(10^{2.3026}\right)^{\text{temp}} = 10^{2.3026 \cdot \text{temp}} $$

(2.12)

The theoretical gain function is then evaluated using these values of $f$. When the logarithm of the independent variable is taken to create the logarithmic scale, the plotted points will in fact be equally spaced on the logarithmic scale.

Having thus determined suitable values of $f$, we evaluate the theoretical gain and overplot this function on the existing graph with the statements

```idl
IDL> r = 15.0e3 Set value of $R$.
IDL> c = 0.0442e-6 Set value of $C$.
IDL> f0 = 1/(2.0*!pi*r*c) Calculate implied reference frequency.
IDL> temp = findgen(76)/25.0 + 1.0 Create vector with values from 1.0 to 4.0.
IDL> f = exp(2.3026 * temp) Create frequencies from $10^1$ to $10^4$.
IDL> gtheory = 1.0/sqrt(1.0 + f0^2/f^2) Calculate theoretical curve.
IDL> oplot, f, gtheory, linestyle=2, thick=3.0 Overplot theoretical curve.
```

Here, we have used the values $R = 15$ kΩ and $C = 0.0442$ µF and then overplotted the theoretical curve with a dashed linestyle.
Finally, to annotate the graph with a legend reminding us of the meaning of each of its pieces, we would use the statements

```
IDL> plots, [500.0, 900.0], [0.3, 0.3], linestyle = 2, thick=3.0
IDL> xyouts, 1000.0, 0.3, 'Theoretical'
IDL> plots, 700.0, 0.25, psym = 4
IDL> xyouts, 1000.0, 0.24, 'Experimental'
```

The first of these statements uses `plots` to draw a line from data coordinate (500.0, 0.3) to the data coordinate (900.0, 0.3). The second then uses `xyouts` to apply the label “Theoretical” to the line, with the label starting at the point (1000.0, 0.3). The remaining statements place a diamond plotting symbol and label it “Experimental”.

The statements listed above and used to produce Fig. 2.9 have been compiled into the file `plotrc.pro`, which is available where all other files mentioned in this chapter are found.\textsuperscript{38}

\section*{2.11 Making Hard Copy}

\subsection*{2.11.1 \ldots of Text}

To print mathematical results from IDL when IDL is being run on a device with X-window output, it is easiest to use the cutting and pasting capabilities of the X-window system/clipboard to copy the information from the window running IDL to the input window of the text editor in use. Alternatively (and necessarily on devices without X-window capability), the command `journal` described in Section 2.4 will create a file containing everything written to the screen in an IDL session, which file can then be edited to delete the unwanted portions.

\textsuperscript{38}See footnote 23.
2.11.2 ... of Graphs

After a function or an array has been plotted on the screen, we sometimes wish to have a hard copy of the graph. Making that hard copy in IDL requires two steps. First, we must create a file containing a PostScript description of the plot. Then, we must send that file to the printer. The second of these steps is a task for the operating system and is described in the Local Guide. The first step, however, is a task for IDL and is accomplished with the statements

```idl
set_plot, 'ps'
device, filename='filename.ps'
```

Specify PostScript output.
Identify file for output. The file will be created in the default directory.
... Re-execute statements that produced plot on screen
... Close file.
set_plot, 'x'

Create graph, which goes to file, not to screen.
Close file.
Restore X-window output.

Note that some features of a graph as it appears on the screen may not be exactly replicated in the output produced by a PostScript file. In particular the effect of the keywords `charsize` and `thick` in the plot command may change when a new output device is selected. Thus, when these (and perhaps other) keywords are invoked, some experimentation may be necessary to produce a printed graph that exactly reflects what may have been displayed on the screen. Further, be particularly aware that, once IDL has been told to produce output in PostScript format, nothing will be directed to the plot window on the screen—even as statements are being executed in the creation of the file. The last statement in the above sequence restores normal interaction between the program and plot window on the screen.

For example, to obtain a file (in the default directory) containing a PostScript description of the first graph in Section 2.10.1, we would execute the statements

```idl
IDL> set_plot, 'ps'
IDL> device, filename='bfield.ps'
IDL> plot, z, B, title='Magnetic Field on z-axis', $
IDL> xtitle='Dimensionless Position', ytitle='Magnetic Field'
IDL> device, /close
IDL> set_plot, 'x'
```

Note that, here—and even more—when the desired graph requires many statements, we should adopt a strategy of constructing and debugging the graphical statements in a batch file. That file can be edited and re-executed until the on-screen display is as we wish. Then, we simply change the device, define an output file, and re-execute the batch file to produce the PostScript file, saving the labor of and potential errors in retyping the statements that produce the display.

2.12 Graphing Scalar Functions of Two Variables

Suppose the function we wish to explore graphically is a function of two independent variables, for example,

\[ z(x, y) = \sin(2\pi x) \sin(3\pi y) \]  

(2.13)

---

39Since many word processing and publishing packages have the capability to import PostScript files, this file is also useful when the graph is to be incorporated in a document.  
40This statement, of course, supposes that IDL is being run on a device with X-window output. The proper statement to return to screen display in Microsoft Windows is `set_plot, 'win'`; on a Macintosh computer, the statement is `set_plot, 'mac'`. Other less frequently used possibilities are described in the IDL manuals.
which defines the shape of the [2,3] mode of oscillation for a square membrane, or

\[ I(\xi,\eta) = \left( \frac{\sin \xi}{\xi} \right)^2 \left( \frac{\sin \eta}{\eta} \right)^2 \]  

(2.14)

which is related to the intensity in the diffraction pattern produced by a square aperture. In graphing a function of a single variable, we used as input to plot the values of the function at a selected set of (regularly or irregularly spaced) values spanning the desired range of that variable. Here, we need as input to any graphing routine the values of the function at a regular, three-dimensional grid of selected points \((x_{ij}, y_{ij})\) covering the portion of the \(xy\) plane within which we seek to display the function graphically. In the broadest of terms, our task then will involve three steps:

1. Create two arrays, one \((x, x_i, \ldots)\) containing the first coordinate \((x, \xi, \ldots)\) of all the points in the grid and the other \((y, \eta, \ldots)\) containing the second coordinate \((y, \eta, \ldots)\).

2. Evaluate the function over that grid, i.e., generate the array containing values of the dependent variable at the points identified in the arrays containing the independent variables. Because IDL processes arrays element by element, a single statement, which would have the forms

\[ z = \sin(2.0*!pi*x) \cdot \sin(3.0*!pi*y) \]
\[ I = (\sin(xi)/xi)^2 \cdot (\sin(eta)/eta)^2 \]

for the functions in Eqs. (2.13) and (2.14), will usually suffice.

3. Invoke one or another graphical display routine, specifying the array \((z, I, \ldots)\) containing the dependent variable as input and perhaps specifying one or more further arguments to control details of the display.

In this section, we describe how to create the necessary arrays containing values of the independent variables and then illustrate how to create different displays of these functions using surface, contour, shade_surf, xsurface, and show3.

### 2.12.1 A Preliminary: The Procedure lugen_grid

Similar to the IDL procedure findgen but functioning in two dimensions, the Lawrence-written procedure lugen_grid creates a pair of two-dimensional arrays, one containing the \(x\) coordinates of a grid of points uniformly spaced in a region of the \(xy\) plane and the other containing the \(y\) coordinates of points in that grid. This procedure has six arguments. The two “free” arguments name the arrays into which lugen_grid should place values of \(x\) and \(y\); the four other arguments, which are specified by the keywords xrange, nx, yrange, and ny, supply the interval over which each independent variable is to extend and the number of segments into which each interval is to be divided.

Thus, for example, the statement

IDL> lugen_grid, x, xrange=[-3.0, 3.0], nx=3, y, yrange=[-1.0, 1.0], ny=4

creates the arrays

\[ x = \sin(2.0*!pi*x) \cdot \sin(3.0*!pi*y) \]
\[ I = (\sin(xi)/xi)^2 \cdot (\sin(eta)/eta)^2 \]

Note that the division of the \(x\) interval \([-3.0, 3.0]\) into three segments results in four values in the array \((i.e., -3, -1, 1, 3)\). In the same manner, the division of the \(y\) interval \([-1.0, 1.0]\) into four segments results in five values \((-1, -0.5, 0, 0.5, 1)\).
where each entry in a column of x has the same value and each entry in a row of y has the same value. Once these two arrays have been created, we can create an identically sized array containing values of the function with a single statement, say

IDL> z = x^2 + y^2

Here, each entry in the array z is the value of the function z at the point whose x coordinate is at the corresponding position in the array x and whose y coordinate is at the corresponding position in the array y.

To illustrate both the use of lugen_grid and the nature of its output, we above chose an example that is far too small to produce a useful display of any function. To prepare for displaying the functions of Eqs. (2.13) and (2.14), then, let us create rather larger arrays z and I with the statements

IDL> lugen_grid, x, xrange = [ 0.0, 1.0 ], nx = 40, 
IDL> y, yrange = [ 0.0, 1.0 ], ny = 40
IDL> z = sin( 2.0*!pi*x ) * sin( 3.0*!pi*y )
IDL> lugen_grid, xi, xrange=[-3.0*!pi, 3.0*!pi], nx=49, 
IDL> eta, yrange=[-3.0*!pi, 3.0*!pi], ny=49
IDL> I = (sin(xi)/xi)^2 * (sin(eta)/eta)^2

(The use of 49 divisions—rather than 50—in the third statement is designed to avoid the appearance of the explicit value 0 in either xi or eta, thereby avoiding several divisions by zero in evaluating I.\footnote{IDL actually handles division by zero in a sensible way. Rather than terminate execution, IDL returns the value \texttt{Inf} and goes on to the next statement presented to it. Since the command \texttt{plot}, for example, is able to ignore \texttt{Inf}, division by zero normally has no more significant impact than the creation of a gap in the resulting graph.})

\subsection*{2.12.2 Surface Plots: The Procedure \texttt{surface}}

The simplest IDL procedure for displaying two-dimensional arrays is \texttt{surface}, and the statements

\begin{verbatim}
IDL> surface, z
IDL> surface, I
\end{verbatim}
2.12. GRAPHING SCALAR FUNCTIONS OF TWO VARIABLES

Figure 2.10: A surface representation of the function $z$.

Figure 2.11: A surface representation of the function $I$.

IDL> surface, z, x, y, xtitle='x', ytitle='y', ztitle='z', title='[2,3] Mode', charsize=2.0
IDL> surface, I, xi, eta, xtitle='!4n!3', ytitle='!4g!3', ztitle='I', title='Diffraction!C!C (Square!C!C Aperture)', charsize=2.0

This will generate a three-dimensional wire-mesh surface conveying the altitudes in the two-dimensional arrays $z$ and $I$. The specification of additional arguments and keywords, however, will create a more refined display. Executing the more elaborate statements\footnote{In the second of these statements, we have invoked IDL's capability to change fonts (e.g., !4 and !3 in xtitle and ytitle) and start a new line (!C in title). These features will be more fully described in Section 2.16.1.}

IDL> surface, z, x, y, az = -40, xtitle='x', ytitle='y', ztitle='z', $\quad$ title='Diffraction!C!C(Square!C!C Aperture)', charsize=2.0

will generate a three-dimensional wire-mesh surface conveying the altitudes in the two-dimensional arrays $z$ and $I$. The specification of additional arguments and keywords, however, will create a more refined display. Executing the more elaborate statements\footnote{In the second of these statements, we have invoked IDL's capability to change fonts (e.g., !4 and !3 in xtitle and ytitle) and start a new line (!C in title). These features will be more fully described in Section 2.16.1.}

IDL> surface, z, x, y, xtitle='x', ytitle='y', ztitle='z', $\quad$ title='[2,3] Mode', charsize=2.0
IDL> surface, I, xi, eta, $\quad$ xtitle='!4n!3', ytitle='!4g!3', ztitle='I', $\quad$ title='Diffraction!C!C (Square!C!C Aperture)', charsize=2.0

for example, will embellish the simple displays by labeling the axes and titling the graph with characters that are a bit larger than the default characters. Here, the second and third “free” (and optional) arguments ($x$ or $xi$; $y$ or $eta$) tell IDL to use the elements of these arrays to scale the two axes. Thus, the axes will be labeled to show a range of $0.0 \leq x, y \leq 1.0$ or $-3\pi < \xi, \eta < 3\pi$. The more complete graphs produced by these last statements are shown in Figs. 2.10 and 2.11. Note that, in the perspective shown, the $x$ and $\xi$ axes run positively to the east northeast, the $y$ and $\eta$ axes run positively to the west northwest, and the $z$ and $I$ axes run positively to the north (vertical).

The procedure \texttt{surface} also admits two optional keywords that control the angle from which the surface is viewed. The keyword \texttt{ax} specifies a rotation, in degrees, about the $x$ axis and the keyword \texttt{az} specifies a rotation, also in degrees, about the $z$ axis. The default values are $ax=az=+30^\circ$. To change the rotation about the $z$ axis to $-40^\circ$, we would use the statements

IDL> surface, z, x, y, az = -40, xtitle='x', ytitle='y', ztitle='z', $\quad$ title='[2,3] Mode', charsize=2.0
IDL> surface, I, xi, eta, az = -40, $\quad$ xtitle='!4n!3', ytitle='!4g!3', ztitle='I', $\quad$ title='Diffraction!C!C(Square Aperture)', charsize=2.0

Figures 2.12 and 2.13 show the resulting rotated views for comparison with Fig. 2.10 and Fig. 2.11.
Examination of Figs. 2.11 and 2.13 suggests that the diffraction pattern might have weak peaks (bright spots) in areas outside the main central peak. Autoscaling of the vertical ($I$) axis, however, is influenced by the high central peak. To show the secondary peaks more clearly, we could make use of yet another keyword to specify a vertical scale that ignores the central peak. For example, the statement

```idl
IDL> surface, I, xi, eta, zrange=[0.0,0.1], 
IDL> xtitle='!4n!3', ytitle='!4g!3', ztitle='I', 
IDL> title='Diffraction!C!C(Square!C!C Aperture)', charsize=2.0
```

which forces the $I$ axis to be confined to the range $0.0 \leq I \leq 0.1$ (and the central peak simply goes off scale). The resulting display, shown in Fig. 2.14, reveals the secondary peaks in the diffraction pattern and even hints at some of those not on the main axes.

Other keywords that can be used with the `surface` procedure are described in the IDL manuals.

### 2.12.3 Contour Plots: The Procedure `contour`

The procedure `contour` draws contour lines for two-dimensional arrays. The general syntax of the `contour` command is similar to that of the `surface` command. Using all of the defaults, we would produce a contour map for the [2,3]-mode of a rectangular membrane with the statement

```idl
IDL> contour, z
```

Exploiting a few keywords, however, we could produce a more meaningful contour map with the statement

```idl
IDL> contour, z, x, y, charsize=1.25, xtitle='x', ytitle='y', nlevels=16
```

where, as with `surface`, the second and third “free” arguments define the scales for the two axes. The keyword `nlevels` overrides the default drawing of contour lines at six values, here stipulating
2.12. GRAPHING SCALAR FUNCTIONS OF TWO VARIABLES

Figure 2.14: A rescaled surface representation of the function $I$.

16 values instead. Alternatively, we could use the keyword levels to stipulate the values at which contour lines are to be drawn, the keyword c_linestyle to stipulate the linestyle to be used for each contour line, and the keyword c_labels to stipulate which contour lines should be labeled with their values. The statement

IDL> contour, z, x, y, charsize=1.25, xtitle='x', ytitle='y', $ 
IDL> levels = [-0.8,-0.6,-0.4,-0.2,0.2,0.4,0.6,0.8], $ 
IDL> c_linestyle = [ 2, 2, 2, 2, 0, 0, 0, 0 ], $ 
IDL> c_labels = [ 0, 1, 0, 1, 1, 0, 1, 0 ], $ 
IDL> thick=3.0

produces the display shown in Fig. 2.15. Facilities available in IDL to make the aspect ratio in this figure 1:1 are described in Section 2.16.5. Details about other keywords that affect the command contour can be found in the IDL manuals.

2.12.4 Shaded Surfaces: The Procedure shade_surf

The procedure surface displays a two-dimensional array using a “wire-mesh” technique in which data points are connected to adjacent points using lines. IDL can also generate an alternate display in which a smooth but shaded surface conveys the values in a two-dimensional array. The procedure shade_surf uses the same syntax as surface and accepts essentially all of the keywords available to surface. To produce shaded surface representations of our two examples, we might execute the statements

IDL> device, decomposed=0
IDL> loadct, 3
IDL> shade_surf, z, x, y, charsize=1.5, $ 
IDL> xtitle='x', ytitle='y', ztitle='z', $

47Behind the scenes, shaded surfaces require fairly large byte arrays. Because of limitations on the sizes of arrays, only quite small shaded surfaces can be produced with the student version of IDL.
IDL> title='[3,2] Mode'
IDL> shade_surf, I, xi, eta, charsize=1.5, 
IDL> xtitle='!4n!3', ytitle='!4g!3', ztitle='z', 
IDL> title='Diffraction!C!C(Square Aperture)'

The statement `loadct, 3` loads a color table (specifically, color table 3; the default is color table 0, which defines a grey scale) for the shaded surface display. Figures 2.16 and 2.17 show these plots, though the printed version using a grey scale pales by comparison with the color plot on the screen. Note that execution of the statement `loadct` will not affect any displays already on the screen and must precede the plotting statements that are to use the new color table.

### 2.12.5 The Procedure `xsurface`

Exploration of the nature of a surface in two dimensions is greatly facilitated by the widget-based procedure `xsurface`. Statements like

IDL> `xsurface, z`
IDL> `xsurface, I`

will bring up a new window that includes not only a display of the surface produced with `surface` but also several buttons and widgets that allow a variety of adjustments in the size, angle of view, and nature of the display. Further, and of particular usefulness, the window contains a box in which the statement that will produce the current display is presented. Thus, we can use this tool to identify the detailed statement to place in a developing pro-file so as to produce the most effective display of the surface.

---

48 The command `loadct` with no arguments will produce a list of the possible color tables and prompt for the entry of the desired number.
2.12.6 The Procedure show3

A final way to represent two-dimensional arrays uses the procedure show3. In a single three-dimensional box, show3 displays a wire mesh surface, a contour plot on the top of the box, and a shaded contour on the bottom of the box. The statements

IDL> show3, z
IDL> show3, I

will produce this composite display for each of our sample functions. The procedure show3 accepts only a few keywords, one of which stipulates keywords to be passed to surface and a second of which stipulates keywords to be passed to contour. The details of structuring more elaborate statements using show3 are fully described in the IDL manuals.

2.12.7 Functions of Two Variables in Polar Coordinates

IDL also makes available the function polar_surface and the procedure polar_contour for plotting functions of the radial and azimuthal polar variables rather than of Cartesian variables. The function polar_surface takes as input

- a two-dimensional array containing values of the surface at a grid of points spanning the region $0 \leq r \leq a$, $0 \leq \phi \leq 2\pi$,
- a vector containing the values of $r$ in that interval, and
- a vector containing the values of $\theta$ (in radians) in that interval

and returns as output a two-dimensional array giving values of the surface on a Cartesian grid of points spanning $-a \leq x, y \leq a$. This latter array can then be displayed using the routines already described for displaying surfaces expressed in Cartesian coordinates. For example, the statements

---

49 We have here placed the arguments to lugen_grid in a different order than that used previously. Remember that IDL distinguishes between “free” arguments and arguments labeled with keywords. The free arguments must appear in the right order, since their position in that order communicates their significance. Keywords can be presented in any order, and they can be mixed in any way with “free” arguments. IDL will not be confused.
IDL> lugen_grid, r, phi, xrange=[0.0,5.0], nx=25, yrange=[0.0,360.0], ny=36
IDL> phi_rad = 2.0*pi*phi/360.0
IDL> x = findgen(51)/5.0 - 5.0
IDL> z = (1.0 - r^2)*exp(-r)*cos(phi_rad)

create arrays r and phi_rad conveying a grid of points in the intervals $0.0 \leq r \leq 5.0$ and $0 \leq \phi \leq 2\pi$ (in radians), create a vector x containing the values of x and y that polar_surface will use,\(^{50}\) and then evaluate the function

$$z(r,\phi) = (1 - r^2)e^{-r} \cos \phi$$  \hspace{1cm} (2.15)

over that grid. With these variables now ready, the single statement

IDL> surface, polar_surface( z, r[*,0], phi_rad[0,*], /grid ), x, x, charsize=1.5

exploits polar_surface to convert the values of z from a polar grid to a Cartesian grid and plot a mesh surface conveying that function. A small number of keywords to polar_surface allows some control over the relationship between the Cartesian and polar grids. When set to true, the keyword grid informs the procedure that the radial and azimuthal grids are regular (points uniformly spaced); another keyword (quintic) allows selection of a slower but more accurate method of interpolating in the conversion. Still others are identified in the IDL manuals. The resulting graph is shown in Fig. 2.18.

The procedure polar_contour takes the same input as polar_surface (though it expects the “free” variables in a different order). As output, polar_contour produces a contour map directly from the polar representation of the surface; no conversion to a Cartesian representation is necessary. For example, given the arrays already evaluated in the previous paragraph, the single statement

IDL> polar_contour, z, phi_rad, r, nlevels=20, thick=3.0, /isotropic

produces the contour map shown in Fig. 2.19. Available keywords, which are fully described in the IDL manuals, include almost all those accepted by the procedure contour. In particular, the keyword isotropic forces equal scaling on both axes.

The purpose of this brief section is to draw attention to the existence of these two routines. Details of their use can be found in the IDL manuals.

2.13 Graphing Scalar Functions of Three Variables

Functions of three variables also occur regularly in physics. Displaying them graphically, however, is complicated because there are four quantities (three independent variables; one dependent variable) to be conveyed. The task involves developing ways to represent four-dimensions in three and then, even more, to project those three onto the two-dimensional screen of a computer workstation. However that is accomplished, the input to the process will have to be a three-dimensional matrix, the $ijk$ element of which gives the value of the function at the point $(x_{ijk}, y_{ijk}, z_{ijk})$ in a regular, three-dimensional grid of points spanning the region of space within which we want to examine the function.

For the sake of a more specific example, 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. In Cartesian coordinates $(x,y,z)$ with the nucleus located at the origin, this probability density is given by the expression

$$p(x, y, z) = \frac{1}{2\pi^3} \frac{e^{-2\rho/3}}{\left(\frac{3z^2}{\rho^2} - 1\right)^2} \frac{1}{2\pi^2(27)^3} e^{-2\rho/3} (9z^4 - 6z^2\rho^2 + \rho^4) \hspace{1cm} (2.16)$$

50By default, the routine will divide the interval $-5.0 \leq x, y \leq 5.0$ into 50 segments.
Figure 2.18: Mesh representation of the function $z$ in Eq. (2.15).

Figure 2.19: Contour map of the function $z$ in Eq. (2.15).

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

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

To be explicit, we determine values of $p(x,y,z)$ over the region $-10.0 \leq x, y, z \leq 10.0$, dividing each axis into 30 segments, which will entail evaluating $p(x,y,z)$ at 31 values of $x$, 31 values of $y$, and 31 values of $z$—a total of $31 \times 31 \times 31 = 29791$ values.

Using IDL to generate a three-dimensional array containing values of this probability density at a grid of points in the desired region is straightforward, since we built `lugen_grid` so that it would also provide the arrays for the independent variables in three dimensions. The array representing the probability density in Eq. (2.16) can be created using the statements

```idl
IDL> lugen_grid, x, xrange=[-10.0, 10.0], nx=30, $  
IDL> y, yrange=[-10.0, 10.0], ny=30, $  
IDL> z, zrange=[-10.0, 10.0], nz=30

IDL> pfactor = 1.0/(2.0*!pi*27.0^3)
IDL> zs = z^2
IDL> rhos = x^2 + y^2 + zs
IDL> rho = sqrt(rhos)
IDL> p = pfactor * exp(-2.0*rho/3.0) * $  
IDL> (9.0*zs^2 - 6.0*zs*rhos + rhos^2)
```

The first statement—which appears distributed over three physical lines—produces a grid of values for the evaluation of the scalar field.\footnote{We here use the optional arguments to `lugen_grid` for the first time. These arguments specify ranges on the $z$ coordinate. Their presence triggers the creation of three-dimensional arrays containing the $x$, $y$, and $z$ coordinates of a grid of points in three dimensions.} The remaining statements evaluate several intermediate quantities and then evaluate $p$, which is a three-dimensional array that will be the input for the production of a variety of displays.

### 2.13.1 Reduction to Two-Dimensional Displays

One way to show the characteristics of a function of three variables is to display two-dimensional subsets of the three-dimensional data, that is, to examine the behavior of the function in planes that...
intersect the volume. The simplest planes to extract are those for which $z$ has a fixed value, since
the expression $p[*,*,5]$, for example, will extract a two-dimensional array containing the values of
$p(x,y,z)$ for which the third coordinate is five (or—better—whatever value of $z$ corresponds to the
integer 5 in the original scaling, $z = z_0 + 5 \times dz$). Thus, the statements\footnote{We must extract two-dimensional arrays from all three arguments, since these statements will not behave properly
with three-dimensional arrays in any position.}

\begin{verbatim}
IDL> surface, p[*,*,5], x[*,*,5], y[*,*,5]
IDL> contour, p[*,*,5], x[*,*,5], y[*,*,5]
IDL> shade_surf, p[*,*,5], x[*,*,5], y[*,*,5]
\end{verbatim}

will produce mesh surface, contour, and shaded surface representations of the probability density in
the plane parallel to the $xy$ plane at the specified value of $z$.

Extracting data in a plane perpendicular to the $x$ or $y$ axes is more complicated, since the
expression $p[15,*,*]$ will not achieve the desired end. Instead, we must be more deliberate. The
statements

\begin{verbatim}
IDL> px = fltarr(31,31)
IDL> for i = 0 , 30 do $
IDL> for j = 0 , 30 do $
IDL> px[i,j] = p[15,i,j]
\end{verbatim}

for example, will extract data in a plane parallel to the $yz$ plane. Then, the statements

\begin{verbatim}
IDL> xx = findgen(31)/1.5 - 10.0
IDL> surface, px, xx, xx, charsize=1.5, $
IDL> xtitle='x', ytitle='y'
IDL> contour, px, xx, xx, nlevels=20, $
IDL> xtitle='x', ytitle='y', /isotropic
\end{verbatim}

will prepare a vector for labeling the axes and generate both a mesh surface and a contour map
representing the probability density in the chosen plane. The figures produced by these commands
are presented in Fig. 2.20 and Fig. 2.21.

While quick, the method of the previous paragraph is restricted to planes in which one has
data, to planes perpendicular to one or another of the coordinate axes, and to displaying only a very
small portion of the data. IDL’s widget-based tool slicer3 is more powerful, since it allows easy
relocation of the plane that “slices” through the three-dimensional volume and also permits oblique
slices. Data are supplied to slicer3 not by passing a variable name but by specifying a pointer as
an argument. When the data to be displayed already exist in IDL’s workspace, we must first create
a pointer to the data with the statement\footnote{The statement *ptr_p = ptr_new( p, /no_copy ) would be less consumptive of memory, and may sometimes be
necessary. Unfortunately, the more elaborate statement will render the original variable p undefined.}

\begin{verbatim}
IDL> ptr_p = ptr_new( p )
\end{verbatim}

Then, we invoke slicer3 with the statement

\begin{verbatim}
IDL> slicer3, ptr_p
\end{verbatim}

A new window containing three menus (FILE and TOOLS along its top; MODE in its control panel),
a number of buttons, and a graph window will be created. As displayed initially, the tool is in ‘slice’
mode and is set to draw orthogonal sections perpendicular to the $x$ axis, though the orientation of
the cut-plane can be changed by buttons labeled ‘X’, ‘Y’, and ‘Z’ in the control panel. After selecting
the orientation of the cut plane, we place the cursor in the larger graph window, drag the plane to
Figure 2.20: Mesh representation of the function $p$ in Eq. (2.16) in the plane $x = x_0 + 15 \times dx = -10.0 + 15 \times 20.0/32.0 = -0.625$.

Figure 2.21: Contour map of the function $p$ in Eq. (2.16) in the plane $x = x_0 + 15 \times dx = -10.0 + 15 \times 20.0/32.0 = -0.625$.

the desired position with ML and, finally, release ML to trigger the drawing of the section. Further sections can be selected in the same way and will be either merged with ('draw' button) or displayed on top of ('expose' button) all previously drawn sections. If the ‘oblique’ button is selected, then ML can be used to drag the normal to the cut-plane into the desired direction and a click of ML on the ‘Display’ button will trigger the creation of the section. To erase the display and start over, we select ‘erase’ from the TOOLS menu. Many other features of slicer3 are described in the IDL manuals. Note particularly that the value of $\text{ptr}_{.p}$ is destroyed on exit from slicer3 and must be recreated before re-entering slicer3.
CHAPTER 2. INTRODUCTION TO IDL

Figure 2.22: An isosurface of the probability density in the state \([n, l, m] = [3, 2, 0]\) using \texttt{isosurface}.

2.13.2 Isosurfaces

Another method of viewing three-dimensional scalar fields is to look at isosurfaces, or sets of locations at which the function has a fixed—and specified—value. In our present example, we locate and display a surface of constant probability density with the statements

\begin{verbatim}
IDL> isosurface, p, 1.2e-4, vertices, polygons
IDL> scale3, xrange=[0.0,30.0], yrange=[0.0,30.0], $
IDL> zrange=[0.0,30.0]
IDL> !p.background=255
IDL> img = polyshade( vertices, polygons, /t3d )
IDL> tvscl, img
\end{verbatim}

Find isosurface.
Set transformation matrices.
Set white background, which must occur prior to the invocation of \texttt{polyshade}.
Create image.
Display the image.

The first statement—\texttt{isosurface}—searches the array \(p\) to generate in \texttt{vertices} and \texttt{polygons} lists of the vertices and polygons that together define a three-dimensional isosurface at which \(p(x,y,z)\) has the value \(1.2 \times 10^{-4}\). The second statement—\texttt{scale3}—converts the specified ranges of the data into the appropriate transformation matrices for projection of the three-dimensional image into the two-dimensional computer screen. \footnote{Other values could be used. We should, of course, be familiar with the range of the data so that the chosen value actually occurs within the data. IDL’s functions \texttt{max} and \texttt{min} can be used to determine those extreme values.} The third statement—\texttt{polyshade}—uses the information in \texttt{vertices}, \texttt{polygons}, and—because the keyword \texttt{t3d} is set—the transformation matrices to construct an image of the isosurface, and the fourth statement—\texttt{tvscl}—displays that image. The result is shown in Fig. 2.22.

The widget-based procedure \texttt{slicer3} also has options that permit the drawing of isosurfaces. We bring up the widget again with the statements

\begin{verbatim}
IDL> ptr_p = ptr_new( p )
IDL> slicer3, ptr_p
\end{verbatim}
\footnote{By setting more parameters in \texttt{scale3}, such as angles, we can display different views of the image.}

The result is shown in Fig. 2.22.
Then, we select ‘surface’ from the Mode menu, select the value at which the isosurface is to be drawn by dragging the vertical line in the small window (which shows a histogram of the values in the database) with any mouse button or entering a value explicitly in the text-entry window below that histogram and, finally, clicking ML on the ‘Display’ button.

An even more useful approach to examining the character of a three-dimensional isosurface exploits the IDL routine `xobjview`. After the quantities `vertices` and `polygons` have been created in the above dialogue, we replace calls to `scale3`, `polyshade`, and `tvscl` with an invocation of `obj_new` to create a suitable object from the data returned by `isosurface` followed by an invocation of `xobjview` to create a view of that object in a window detached from the IDL command window. Specifically, the statements\(^{56}\)

```idl
IDL> wavefct = obj_new('IDLgrPolygon', vertices, $
IDL>   polygons=polygons, color=[255,0,0])
IDL> xobjview, wavefct
```

achieve the desired end. (Here, `IDLgrPolygon` is the reserved name for the specific class of object we need here to create.) Once the `Xobjview` window has been created and the object has been drawn in it, we can use the mouse and the window’s buttons and menus to rotate, resize, or translate the object, to zoom in or out, to restore the original view, and perform many other adjustments—all in real time. Keywords allowed in the arguments of `obj_new` are influenced by the particular class of object to be created, and there are many keywords for each class; here we have illustrated two (`polygons`, which is essential to the creation of the desired object, and `color` (default black), which uses the RGB designation to specify the color for drawing the object). Keywords allowed in the arguments of `xobjview` include `scale` (default \(1/\sqrt{3}\)), `title` (default null), and `[x,y]size` (default 400 pixels), which specify the initial scaling, a title for the display, and the size of the display window. Numerous other keywords are identified and described in the IDL manuals.

### 2.13.3 Other IDL Routines

Additional IDL routines for examining functions of three variables include

- `extract_slice`, which passes a plane of user-specified position and orientation through the three-dimensional data set and constructs a two-dimensional array containing values from which the character of the data in that two-dimensional slice can be examined.

- `project_vol`, which converts a three-dimensional array of values, e.g., the arrays representing scalar functions of three variables, into a two-dimensional array by projecting the data in a way similar to that of an x-ray picture. The value at each point in the two-dimensional array conveys information about the nature of the three-dimensional data along a particular path from the source of the x-ray through the data to the viewing screen.

- `voxel_proj`, whose function is similar to that of `project_vol`, except that the variety of projections is smaller and execution time is therefore likely to be faster.

- `xvolume`, whose function is similar to that of `xsurface` except that it supports interactive manipulation of isosurfaces in three-dimensional scalar fields.

Details on all four of these routines are described in the IDL manuals.

\(^{56}\)We are here simply identifying what is, in fact, an enormous topic: IDL’s capabilities for creating, processing, and displaying objects. Even though we shall in one or two places open a window into the topic, even a superficial treatment would go well beyond the scope of this book.
2.14 Graphing Vector Fields

Graphical display of vector functions of two or three variables is more complicated than display of scalar functions, but IDL has several routines designed to produce such displays. To illustrate the general procedures briefly and without much detail, let us consider the electromagnetic field in a transverse electric (TE) wave in the rectangular waveguide shown in Fig. 2.23 when the electric field has only a $z$ component, the magnetic field has only $x$ and $y$ components, and none of these components depends on $z$. In this case, the only non-zero components of the electric field $\mathbf{E}$ and the magnetic field $\mathbf{H}$ are given at time $t = 0$ by

$$
\begin{align*}
E_z &= \cos \kappa_x b \pi \sin n \pi \gamma \\
H_x &= \sin \kappa_x b \pi \cos n \pi \gamma \\
H_y &= \frac{\kappa_x b}{n \pi} \cos \kappa_x b \pi \sin n \pi \gamma
\end{align*}
$$

(2.18)

where the fields are measured in units in which the amplitude of $E_z$ is one. Further, $b$ is the $y$-dimension of the guide, $n$ is a positive integer, $\xi = x/b$ and $\gamma = y/b$ are dimensionless coordinates within the guide, and

$$
\left(\frac{\kappa_x b}{n \pi}\right)^2 = \left(\frac{\omega b}{n \pi c}\right)^2 - 1
$$

(2.19)

where $c$ is the speed of light and $\omega$ is the frequency of the wave. The field depends on two parameters, $n$ and $\kappa_x b$, which $\kappa_x b$ is determined from Eq. (2.19) by $\omega$, $b$, $c$, and $n$. Since $\kappa_x b$ must be real if the wave is to propagate down the guide, we must require that $(\omega b/n \pi c)^2 > 1$ or that $\omega > n \pi c/b$. 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

$$
\begin{align*}
E_z &= \cos 2 \pi \xi \sin 2 \pi \gamma \\
H_x &= \sin 2 \pi \xi \cos 2 \pi \gamma \\
H_y &= -\cos 2 \pi \xi \sin 2 \pi \gamma
\end{align*}
$$

(2.20)

where $\xi$—the coordinate along the guide can range over any values—we choose $0 \leq \xi \leq 1$—but, to be inside the guide, $\gamma$ is confined to the region $0 \leq \gamma \leq 1$.

For a discussion of the electromagnetic fields in wave guides, see The Theory of the Electromagnetic Field by David M. Cook (Prentice-Hall, Englewood Cliffs, NJ, 1975) or Introduction to Electrodynamics, Third Edition, by David J. Griffiths (Prentice-Hall, Upper Saddle River, NJ, 1999). The first of these books, out of print since the early 1990’s, was available in a Dover reprint after January, 2003, but that edition is also now out of print.
Each component of these fields can now be represented by a two-dimensional array. The \( \textbf{H} \) field, which has two non-zero components, then is translated into two such arrays; the \( \textbf{E} \) field, which has only one non-zero component, requires only one such array. These arrays are readily created with the IDL statements\(^{38}\)

\[
\text{IDL} > \text{lugen} \_\text{grid}, \text{x, y, xrange=[0.0,1.0], nx=30, yrange=[0.0,1.0], ny=30}
\]
\[
\text{IDL} > \text{xp} = 2.0*!\pi*x \quad \text{&} \quad \text{yp} = 2.0*!\pi*y
\]
\[
\text{IDL} > \text{Ez} = \cos(xp) * \sin(yp)
\]
\[
\text{IDL} > \text{Hx} = \sin(xp) * \cos(yp)
\]
\[
\text{IDL} > \text{Hy} = -\cos(xp) * \sin(yp)
\]

### 2.14.1 The Procedure \texttt{velovect}

The IDL procedure \texttt{velovect} produces a two-dimensional vector field plot. At each grid point \texttt{velovect} draws an arrow which conveys both the direction and the magnitude of the field at that point. The calling sequence for this procedure has the form\(^{59}\)

\[
\text{velovect, u, v, x, y}
\]

where \( u \) and \( v \) are the \( x \) and \( y \) components of the two-dimensional field, respectively, and \( x \) and \( y \) (which are optional) are \emph{vectors} containing the actual coordinates of the grid points on the two axes. Thus, for example, the two components of \( \textbf{H} \) in IDL’s memory can be displayed in this form by executing the statement

\[
\text{IDL} > \text{velovect, Hx, Hy, x[*,0], y[0,*], xtitle='x', ytitle='y'}
\]

(Here, \( x[*,0] \) and \( y[0,*] \) extract the first row and first column, respectively, from the two-dimensional arrays \( x \) and \( y \) to create the vectors required by \texttt{velovect}.\) The resulting display is shown in Fig. 2.24.

As with most IDL commands, \texttt{velovect} can be embellished with many keywords. We have already used \texttt{xtitle} and \texttt{ytitle}. Among the others are

- \texttt{length} (default 1.0), which specifies the length of the longest vector as a multiple of the length of a cell, i.e., the distance from one grid point to the next.
- \texttt{color}, which specifies the color index used for the plot.

and all of the keywords accepted by \texttt{plot}.

### 2.14.2 The Procedure \texttt{vel}

Invoked with a statement of the form

\[
\text{vel, u, v}
\]

where \( u \) and \( v \) again contain the \( x \) and \( y \) components of the field,\(^{60}\) the procedure \texttt{vel} draws a vector field with \emph{curved} arrows that follow the field and are proportional in length to the field strength. Each arrow is composed of a number of short segments and is positioned at a random point on the \( xy \) grid. Thus, the statement

\[^{38}\text{From here on in this section, we drop the overbars.}\]
\[^{59}\text{This statement merely illustrates the syntax. Since } u \text{ and } v \text{ have not yet been defined, this specific statement is not itself executable.}\]
\[^{60}\text{The procedure } \texttt{vel} \text{ does not accept additional “free” arguments to specify actual coordinates nor does it accept the keywords } \texttt{xtitle} \text{ and } \texttt{ytitle}.\]
Figure 2.24: Magnetic field at \( t = 0 \) in a rectangular wave guide using `velovect`. The illustrated structure, of course, propagates along the guide as time unfolds.

\[ \text{IDL> vel, Hx, Hy, nvecs=500,} \]
\[ \text{IDL> title = 'Magnetic field in rectangular wave guide'} \]

which uses the keyword `nvecs` to set the number of vectors drawn to 500 (default 200) and the keyword `title` to specify a title (default ‘Velocity Field’), produces the plot shown in Fig. 2.25. Other useful keywords include

- `length` (default 0.1), which specifies the length of each line segment used to draw the curved arrows. It is expressed as a fraction of the longest vector divided by the number of steps.
- `nsteps` (default 10), which specifies the number of line segments used to draw each arrow.

### 2.14.3 The Procedure `plot_field`

The procedure `plot_field` also plots a two-dimensional field with a statement of the form

\[ \text{plot_field, u, v} \]

where \( u \) and \( v \) are defined above. This procedure places each arrow at randomly picked points on the plane. In addition to the keyword `title` (default ‘Velocity Field’) as described for the `vel` procedure, some of the keywords accepted by this command are

- `n` (default 200), which specifies the number of arrows to be drawn.
- `length` (default 0.1), which specifies the length of the longest vector as a fraction of the plotting area.

The display produced by `plot_field` is similar to that produced by `vel` and is not shown.
2.14.4 More Elaborate Displays

Numerous additional displays can be created. We illustrate the possibilities with a plot in which the $x$ and $y$ components of the magnetic field displayed with `velovect` are superimposed on a contour map of the $z$ component of the electric field. The statements

```idl
IDL> velovect, Hx, Hy, x[*,0], y[0,*], xtitle='x', ytitle='y'
IDL> contour, /overplot, Ez, x, y, thick=3.0, $
IDL> levels= [-0.9,-0.7,-0.5,-0.3,-0.1,0.1,0.3,0.5,0.7,0.9], $
IDL> c_linestyle=[ 2, 2, 2, 2, 2, 0, 0, 0, 0, 0]
```

produce the display shown in Fig. 2.26. Here, we have used the keywords `levels` and `c_linestyle` to specify, first, that contours at particular values are to be drawn and, second, that contours for negative values should be drawn with a dashed line (linestyle 2) while contours for positive values should be drawn with a solid line (linestyle 0). Thus, the electric field is coming out of the page in regions where its component is indicated with a solid line and going into the page in regions where its component is indicated with a dashed line.

2.14.5 Three-Dimensional Vector Fields

When a vector field has non-zero components in more than two coordinate directions, a picture of the field is much harder to draw and even more difficult to interpret without the aid of a tool that will allow easy viewing of the picture from a variety of viewing angles. One can, of course, use the procedures already discussed to draw pictures of the projections of the field in any given plane into that plane. In addition, IDL provides the procedure `flow3` (see the IDL manuals) for drawing a picture similar to those produced by `vel` and `plot_field`, but drawing three-dimensional space curves to represent each field line. The primary input to `flow3` is a trio of three-dimensional arrays giving the $x$, $y$, and $z$ components of the vector field at a grid of points spaced through the volume of interest. The exploration of these capabilities is left to an exercise.
Figure 2.26: Plot showing both magnetic and electric fields at \( t = 0 \) in wave guide. The magnetic field lines are closed curves in the plane of the paper. The electric field is directed perpendicular to the paper and has at each point a component conveyed by the contour lines, where solid lines indicate a positive component and dashed lines a negative component. The illustrated structure, of course, propagates along the guide as time unfolds.

2.15 Animation

Along with all of the image processing and graphing capabilities, IDL can also display a sequence of images in rapid succession to produce an animated display. In essence, one creates each image in turn in an appropriately sized window, reads each window as a bit-map into an element of a properly sized byte array, and then displays each of those bit-mapped images in turn.\(^{61}\) To illustrate this process, we use a data file containing a succession of 41 \( 33 \times 33 \) arrays, each of which conveys the displacement of a square drumhead at a particular moment in time.\(^{62}\) The statements creating a three-dimensional byte array containing the 41 images in the animation are\(^{63}\)

\[
\begin{align*}
\text{IDL} & > \text{openr, 1, 'ldrum.dat'} \\
\text{IDL} & > \text{disp} = \text{fltarr}(33, 33, 41) \\
\text{IDL} & > \text{readf, 1, disp} \\
\text{IDL} & > \text{close, 1}
\end{align*}
\]

\(^{61}\)Limitations on array sizes in the student version of IDL constrain the extent to which animated displays can be produced with that version.

\(^{62}\)The file was created by using \texttt{lsode}—the Livermore solver for ordinary differential equations—to effect a numerical solution of the wave equation for the drumhead.

\(^{63}\)We assume that the file \texttt{ldrum.dat} is stored somewhere in IDL’s search path. See footnote 23 if that is not the case.
2.15. ANIMATION

IDL> window, /free, title='Animation', $  
Create 600 pixel × 600 pixel window for  
display and make that window active. The  
keyword /free tells IDL to pick a window  
index to avoid conflicts.  

IDL> xsize = 600, ysize = 600  
Create array to store images.  

IDL> frames = bytarr(600,600,41)  
Create i-th image.  

IDL> for i = 0, 40 do begin $  
Read screen.  

IDL> shade_surf, disp[*,*,i], zrange=[-0.25,0.25] & $  
Create i-th image.  

IDL> frames[0,0,i] = tvrd(0,0,600,600) & $  

IDL> endfor

At this point, the array frames contains all of the images stored as bit maps so they can be displayed  
one after the other in quick succession. In this coding, note the following: (1) the byte array frames,  
in which bit-mapped representations of images are stored, requires 600 × 600 × 41 bytes ≈ 14.8 MB  
of memory and (2) the arguments of the command tvrd, which reads the contents of the window  
and stores it in the byte array, give the coordinates of the bottom left and the top right corners of  
the window (in pixels).

Several options for displaying the images can be invoked. The command tvscl, for example,  
displays a single image, scaling the colors to use the full spectrum available on the display device.  
Thus, the statement

IDL> tvscl, frames[*,*,0]  
will display the first frame and the statement

IDL> for i = 0, 40 do tvscl, frames[*,*,i]  
will display all 41 frames in quick succession. In older versions of IDL, we would have now used the  
procedure movie with a statement like

IDL> movie, frames, order=0  
% Compiled module: MOVIE.  
S for slower, F for faster, Q to quit.

which would have displayed each image in the array frames one after another from beginning to  
end, repeatedly until stopped. This route offered control over the rate at which frames are displayed,  
but the procedure movie has been deleted from more recent versions of IDL.

In more recent versions of IDL, we would use a still more elaborate widget-based IDL procedure  
called xinteranimate, which offers substantial interactive control over the animation by allowing  
it easily to be stepped forward or backward frame by frame. Three statements are needed to set it  
up. First, we must initialize the animation tool with a statement like

IDL> xinteranimate, set = [ 600, 600, 41 ], /showload

This statement creates a window titled ‘XInterAnimate’, which in turn includes a display window  
that is 600 × 600 pixels in size and prepares the tool for receiving 41 images of that size. Then, the  
statement

IDL> for i = 0, 40 do xinteranimate, frame = i, image = frames[*,*,i]  
will load the images into the tool, one at a time, and—because the keyword showload was set to  
true in the previous statement—will display each image as it is loaded. Finally, the statement

IDL> xinteranimate
will initiate the animation and make all of the controls active so we can speed up or slow down the animation, pause it, change the direction of its progress, and select specific frames for closer examination. Clicking ML on the ‘End Animation’ button will exit from the animation tool.\footnote{Do not close the window by clicking on the $\times$ in its upper right corner. That action may hang the system in a way from which it will be difficult to recover.}

The procedures described in the previous two paragraphs ultimately display a succession of images, each of which is separately defined from the input data. An alternative procedure is better suited to the task of creating an animation that can be generated by explicit transformation of a single image in some simple way. In this case, only the data defining the initial image need be supplied. IDL procedures can then be used to generate the remaining images from that initial image. For example, generating an animated display showing the simple harmonic motion of a drumhead is especially easy if we have data on its initial shape. One simply creates a succession of images, each generated from the initial image by multiplying all values defining that image by $\cos 2\pi t$, where $t$ is stepped in equal increments from 0 to 1 so the multiplier runs over one period.\footnote{Whether we stop one image short of $t = 1$ or include the image for $t = 1$ depends on how the animation is to be shown. If the display moves forward from the first to the last image and then moves backward from the last to the first, the image for $t = 1$ should be included. If the display moves forward from the first to the last image and then jumps abruptly back to the first image, the image for $t = 1$ should be omitted. We here assume the first mode of display.}

For example, to display the sinusoidal motion of a square drumhead whose initial shape is given by

$$u(x, y, 0) = \sin 2\pi x \sin 3\pi y, \quad 0 \leq x, y \leq 1$$

we might start with the statements

```idl
IDL> lugen_grid, x, y, xrange=[0.0,1.0],$  
  IDL nx=25, yrange=[0.0,1.0],ny=25  
IDL> u = sin(2*!pi*x) * sin(3*!pi*y)  
IDL> t = findgen(41)/40.0  
IDL> disp = fltarr(26,26,41)  
IDL> for i = 0, 40 do $  
IDL>   disp[*,*,i] = u*cos(2*!pi*t(i))
```

At this point, the necessary data are stored in the 3D array `disp`. From here, the creation of the animated display follows exactly the pattern exhibited in the second and third paragraphs of this subsection. We simply have generated the necessary input data by calculating it from the initial shape rather than by reading it from a file.

### 2.16 Advanced Graphing Features

In this section, we describe some of IDL’s more sophisticated capabilities. Because these routines are more involved than those previously described, it would be best to master the basic graphics routines before moving on to these examples.

#### 2.16.1 Fonts

Among the most useful of IDL’s auxiliary features is a capability to choose characters from a wide variety of fonts (including sans serif, italic, and bold characters, Greek letters, and common mathematical symbols) and to position characters as subscripts and superscripts. Unfortunately, the manner in which these features are invoked is awkward and non-intuitive. The example in this subsection is intended to introduce some of IDL’s capabilities but is in no way a substitute for careful reading of IDL’s own documentation on the topic.

Fonts and character positions are selected by commands embedded in quoted strings. Each such command begins with an exclamation point `!” and is completed with a code, some of which are
Table 2.4: Some codes for use in character strings in IDL. Codes in the first and third columns identify fonts. Codes in the fifth column effect changes in position and size of the characters drawn. The simplex Roman font (!3) and the normal level (!N) are the defaults.

<table>
<thead>
<tr>
<th>Code</th>
<th>Font</th>
<th>Code</th>
<th>Font</th>
<th>Code</th>
<th>Shift to</th>
</tr>
</thead>
<tbody>
<tr>
<td>!3</td>
<td>Simplex Roman</td>
<td>!12</td>
<td>Simplex Script</td>
<td>!C</td>
<td>New line</td>
</tr>
<tr>
<td>!4</td>
<td>Simplex Greek</td>
<td>!13</td>
<td>Complex Script</td>
<td>!D</td>
<td>Subscript 0.62 size</td>
</tr>
<tr>
<td>!5</td>
<td>Duplex Roman</td>
<td>!14</td>
<td>Gothic Italian</td>
<td>!E</td>
<td>Superscript 0.44 size</td>
</tr>
<tr>
<td>!6</td>
<td>Complex Roman</td>
<td>!15</td>
<td>Gothic German</td>
<td>!I</td>
<td>Subscript 0.44 size</td>
</tr>
<tr>
<td>!7</td>
<td>Complex Greek</td>
<td>!16</td>
<td>Cyrillic</td>
<td>!N</td>
<td>Normal level</td>
</tr>
<tr>
<td>!8</td>
<td>Complex Italic</td>
<td>!17</td>
<td>Triplex Roman</td>
<td>!U</td>
<td>Superscript 0.62 size</td>
</tr>
<tr>
<td>!9</td>
<td>Math</td>
<td>!18</td>
<td>Triplex Italic</td>
<td>!20</td>
<td>Miscellaneous</td>
</tr>
<tr>
<td>!10</td>
<td>Special</td>
<td>!19</td>
<td>Miscellaneous</td>
<td>!20</td>
<td>Miscellaneous</td>
</tr>
<tr>
<td>!11</td>
<td>Gothic English</td>
<td></td>
<td></td>
<td>!20</td>
<td>Miscellaneous</td>
</tr>
</tbody>
</table>

identified in Table 2.4. For example, a Greek character set is selected with the command !4 while raising characters to the position of a superscript and shrinking their size is effected by one of the commands !E or !U. Returning to the default character set is accomplished with the command !3 and to the original character position by the command !N. Further details on these commands and information about which characters are in which positions in which character sets can be found in the IDL manuals. Further, the statement

IDL> showfont, 3, "Arbitrary Title of User’s Choice"

will create a display of all the characters in font 3. Of course, reserving the exclamation point to introduce commands of this sort makes conscious printing of that character difficult; to include an exclamation point explicitly in the printed output, type two exclamation points "!!" in the character string.

In Section 2.12.2, we have already seen one use of this capability. We present one more example. Let us embellish the graph of a magnetic field first presented in Fig. 2.5 by (1) changing the font used for the various titles and (2) replacing the title on the y axis with the equation itself. The statements

IDL> dz = 8.0/100.0
IDL> z = -4.0 + dz * findgen(101)
IDL> B = (1.0 + z^2)^(-1.5)
IDL> plot, z, B, title='Magnetic Field on !8z!6 Axis',
IDL>                  xtitle='!6Dimensionless Position, !8z!3',
IDL>                  ytitle='!8B!6(!8z!6)=1/(1+!8z!6!E2!N)!E3/2!N!3',
IDL>                  thick=3.0, charsize=1.5, ticklen=1.0

will produce the revised graph of the magnetic field shown in Fig. 2.27. Here, the commands !6 and !8 in a character string select the complex roman and complex italic fonts, respectively. In ytitle, the command !E was used to raise the character to the exponent level and automatically reduce the character size and the command !N was used to return to the normal level and original character size. As always, restoring the font to the default font with !3 is prudent, since changes in font made in one statement will sometimes “bleed” into subsequent statements.

2.16.2 Space Curves

Sometimes it is desirable to view the trajectory of a particle in three-dimensional space. IDL has the ability to take vectors of x, y, and z coordinates, project these points onto the two-dimensional
screen, and then connect consecutive points with lines. To illustrate this feature of IDL, let $\alpha$ be a constant and consider the curve defined parametrically with the equations

$$x = \cos t ; \quad y = \sin t ; \quad z = \alpha t$$

These equations describe the trajectory of a charged particle moving in a constant magnetic field directed along the $z$ axis. The starting point is, of course, to evaluate $x$, $y$, and $z$. The next step is to use the procedure `scale3` to calculate and store the transformation matrix for projecting these three-dimensional coordinates onto the two-dimensional screen. Procedure `scale3` has three mandatory “free” arguments giving the $x$, $y$, and $z$ coordinates of points on the curve and keywords giving the appropriate $x$, $y$, and $z$ ranges. Finally, the procedure `plots` (not to be confused with the plain vanilla procedure `plot`) is used to plot the graph. The statements

```idl
IDL> erase
IDL> t = 30.0*findgen(201)/200.0
IDL> alpha = 0.5
IDL> x = cos(t)
IDL> y = sin(t)
IDL> z = alpha*t
```

will produce the graph in Fig. 2.28. Note in particular that `scale3`, which also allows the keywords

[66]The command `plots` to be used presently does *not* automatically erase the current display. Without this statement, the graph here being set up will ultimately overlay any existing display.
Figure 2.28: Space curve representing the trajectory of a charged particle moving in crossed electric and magnetic fields.

ax and az to override the default rotations about the x and z axes, creates the transformation matrix which is then invoked by the command plots with the inclusion of the keyword /t3d. Note also that a new invocation of plots to produce space curves will not automatically clear the plot window; to erase the plot window prior to calling plots a second time, use the erase command.

An even more useful approach to examining the character of a three-dimensional space curve exploits the IDL routine xplot3d. After, the three coordinates x, y, and z have been evaluated, we simply execute the statement

IDL> xplot3d, x, y, z

and wait for the display to be produced. Once the XPlot3D window has been created and the space curve has been drawn in it, we can use the mouse and the window’s buttons and menus to rotate, resize, or translate the image, to zoom in or out, to change the style of the axes, to restore the original view, and perform many other adjustments—all in real time. Numerous keywords, including thick, [x,y,z]range, title, [x,y,z]title, and linestyle can be included to modify the details of the display.

2.16.3 System Variables

Many of the features of IDL are controlled by the values of system variables, the most important of which for graphical operations are !p, !x, !y, and !z. These variables are structured variables with many components, each of which has its own (sub)structure and each of which has a default value. We have already seen how the system variable !p.multi, which is a component of !p, allows for possible placement of more than one graphic image in a single window. A full listing of the components of these system variables will be found in the IDL manuals. Frequently useful components of these system variables include

- !p.background (default 0), a scalar that sets the background color for the graph window. With the default color table, the values 0 and 255 give black and white backgrounds, respectively.
- !p.charsize (default 1.0), a scalar that sets the size of the characters drawn to annotate a graph.
• \texttt{!p.color} (default 255), a scalar that sets the color for the lines drawn in the graph window. With the default color table, the values of 0 and 255 give black and white lines, respectively. Note that black lines will be invisible on the screen unless the background color is changed away from its black default.

• \texttt{!p.linestyle} (default 0), a scalar that specifies which of several line styles should be used for the drawing of the lines of a graph. Possible values and their effects are listed in Table 2.2.

• \texttt{!p.thick} (default 1), a scalar that controls the width of the lines used to draw graphs (but has no effect on the width of lines used for axes).

• \texttt{!p.psym} (default 0), a scalar that—when different from zero—identifies the plotting symbol to be used by \texttt{plot}, \texttt{oplot}, and \texttt{plots}.

• \texttt{!p.ticklen} (default 0.02), a scalar that specifies the length of the (major) tick marks, expressed as a fraction of the plot size.

• \texttt{!x.range}, \texttt{!y.range}, and \texttt{!z.range} (default for each \([0.0,0.0,0.0]\), which results in autoscaling), two-component vectors that specify the range of values on the \(x\), \(y\), and \(z\) axes, respectively.

• \texttt{!x.style}, \texttt{!y.style}, and \texttt{!z.style} (default for each 0), scalars specifying the style of the axes. Full details, which are complicated, are described in the IDL manuals.

• \texttt{!prompt} (default \texttt{IDL>}), a scalar that identifies the prompt displayed when IDL is waiting for a statement.

Explicit change of the value of a system variable makes the new value “permanent” (i.e., until we change it again—or exit from IDL). Similarly named keywords in many of the graphics commands permit overriding the default values set by the system variables, but only within the statement in which the keyword is specified. Statements use the values of the system variable as they exist when the statement is invoked unless the statement itself contains an explicit stipulation of a keyword overriding the current value of the system variable. The IDL manuals also contain a listing of these keywords and indicate correspondences between keywords and system variables.

2.16.4 Using Multiple Windows

In the simple circumstances so far introduced, we have accepted IDL’s willingness to create a window (IDL 0) when a plot command is invoked or to use an existing window 0 if it had already been created by a previous plot command. We can manipulate additional windows with statements like

\begin{verbatim}
window, n
window, n, title='New Window'
window, n, xsize=400, ysize=400
window, n, xpos=400, ypos=200

window, /free
wdelete, n
wset, n
wshow, n
\end{verbatim}

Create window \(n\), where \(n = 1, 2, \ldots\).
Create window \(n\) with title.
Create window \(n\) with size 400 \(\times\) 400 pixels.
Create window \(n\), overriding default positioning to place lower left corner 400 pixels to the right of and 200 pixels above the lower left corner of the screen.
Create window with an available free index.
Delete window \(n\).
Select window \(n\) as the default.
Bring window \(n\) to foreground.

Note that execution of the statement \texttt{window, 2} (say) when window 2 already exists will delete the existing window before creating the new one. Note also that the system variable \texttt{!d.window}, whose value is \(-1\) when no plot window is on the screen, always contains the index of the current plot window; the commands \texttt{window} and \texttt{wset} manipulate this system variable, which we can print if we forget its value but we should not change it except indirectly through using those commands.
that manipulate it behind the scenes. Plot commands always direct their output to the current plot window. Thus, the statements

```
IDL> x = findgen(101)/10.0 & sine = sin(x) & cosine = cos(x)
IDL> window, 0, title = 'Sine Curve'
IDL> window, 1, title = 'Cosine Curve'
IDL> wset, 0 & plot, x, sine
IDL> wset, 1 & plot, x, cosine
```

will create two separate windows, each with a suitable title, and then plot the sine curve in the first window and the cosine curve in the second. Both will be visible on the screen at the same time; the second plot command will not erase the first window.

### 2.16.5 Scaling so Circles are Circles

If we accept all of the system defaults, the graph produced by the statements

```
IDL> s = findgen(101)/10.0
IDL> x = cos(s) & y = sin(s)
IDL> plot, x, y
```

will not appear at all as the circle it should be. More recent versions of IDL, however, make available the keyword `isotropic`, which is legal with the commands `plot` and `contour` (and perhaps others). When invoked, this keyword forces equal coordinate increments on the two axes to be displayed as equal distances on the axes on the screen. Thus, for example, the statements

```
IDL> s = findgen(101)/10.0
IDL> x = cos(s) & y = sin(s)
IDL> plot, x, y, /isotropic
```

will produce a graph in which the circle appears as a circle.

Unfortunately, while a graph produced in this way and written to a PostScript file will preserve the desired aspect ratio, the graph itself will not be centered in its bounding box and hence, for example, in a center environment in \LaTeX{} will not directly produce a centered graph on the page. One way to address this complication is to wrest control of some further defaults from the program so as to adjust the aspect ratio and control the portion of the window in which the plot appears. We illustrate the necessary adjustments first for on-screen displays. The keywords `xsize` and `ysize` to the `window` command and the system variables `!x.margin` and `!y.margin` are the important features of IDL in this context. Even then, however, we must be careful, since the size parameters for the `window` command specify the size of the entire window, including the title bar. We must begin by making sure that the region of the window available for plotting is square, which means that the vertical extent of the window must in fact be (about\(^{67}\)) 30 pixels larger than the horizontal extent. A statement like

```
IDL> window, 0, xsize=400, ysize=430
```

will achieve the desired end. Then, however, we must also override the default sizes that the `plot` command assumes for the margins around the axes, setting them instead so that the axes will embrace a square region. Those margins are controlled by the system variables `!x.margin` and `!y.margin`, each of which is a two-element vector. We might, for example set them to the values

\(^{67}\)The value 30 was determined empirically by measuring on the screen.
IDL> !x.margin=[ 5.0, 2.0 ]
IDL> !y.margin=[ 3.0, 4.0 ]

(The sum of the two values in the vector assigned to !x.margin must be the same as the sum of the two values in the vector assigned to !y.margin.) With these stipulations, the statements with which this subsection began will now produce a graph that actually looks like a circle on the screen. Similarly, the statements

IDL> window, 0, xsize=400, ysize=430
IDL> !x.margin=[ 5.0, 2.0 ]
IDL> !y.margin=[ 6.0, 1.0 ]
IDL> dt = 2.0*!pi/100.0
IDL> theta = dt*findgen(101)
IDL> a = 3.0
IDL> r = a*(1.0 - cos(theta))
IDL> plot, r, theta, /polar, 
IDL> subtitle='The Cardioid', 
IDL> thick=3.0, charsize=1.25
IDL> axis, 0, 0, xaxis=0
IDL> axis, 0, 0, yaxis=0

will produce a graph of the cardioid in which the two axes have identical scale increments on the two axes so that, for example, one scale unit along the horizontal axis represents the same physical length as one scale unit along the vertical axis.

Communicating this circle to the printer through a PostScript file, however, requires additional specifications. Again, one uses xsize and ysize keywords, this time to the device command, so as to set the size of the plotting area on the paper away from its default size. One also needs the flag /inches to establish the units in which dimensions are specified. Further (though not here illustrated), one may need the keywords xoffset and yoffset to make sure that the entire graph actually appears on the printed page. The statements

IDL> set_plot,'ps'
IDL> device, filename='test.ps', $ 
IDL> xsize=4.0, ysize=4.0*(float(430)/float(400)), /inches 
IDL> plot, x, y
IDL> device,/close
IDL> set_plot,'x'

will set the plotting area to a size that has the same aspect ratio as the display produced on the screen, thus assuring that the printed plot will also appear as a circle. Compare Fig. 2.29, which was produced by the procedures here described, with Fig. 2.7, in the production of which we ignored these issues of scaling.

2.16.6 Working with Color

By default, IDL produces its displays in a grey-scale, using color table 0, which is officially named ‘B-W linear’. The command loadct allows that default to be changed to any of three dozen or more built-in alternative color tables, each of which makes 255 distinct colors available. For example, the statements

device, decomposed=0
loadct, 3
plot, x, y, color=145
will load color table 3 as the current color table and then produce a graph of $y$ versus $x$ using color 145 from that table.\(^\text{68}\) Color table 3 will remain the current table until it is explicitly changed. A list of the available built-in color tables will be displayed if the command `loadct` is issued without an argument. Alternatively, the command `xloadct` will bring up a browser in which we can select any of the available color tables and see the resulting spectrum before making the selection final.

Even more generally, IDL allows us to construct our own correspondence between colors and color numbers. Each color in the spectrum is specified most simply by giving three values that determine the relative proportion of the primary colors red, green, and blue.\(^\text{69}\) For example, red would be represented by the RGB triplet $(255,0,0)$, green by the triplet $(0,255,0)$, and blue by the triplet $(0,0,255)$. The table

<table>
<thead>
<tr>
<th>Color Name:</th>
<th>Black</th>
<th>Blue</th>
<th>Green</th>
<th>Cyan</th>
<th>Red</th>
<th>Magenta</th>
<th>Yellow</th>
<th>White</th>
</tr>
</thead>
<tbody>
<tr>
<td>Color Index:</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>Red value</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>255</td>
<td>255</td>
<td>255</td>
<td>255</td>
<td>255</td>
</tr>
<tr>
<td>Green value</td>
<td>0</td>
<td>0</td>
<td>255</td>
<td>255</td>
<td>0</td>
<td>0</td>
<td>255</td>
<td>255</td>
</tr>
<tr>
<td>Blue value</td>
<td>0</td>
<td>255</td>
<td>0</td>
<td>255</td>
<td>0</td>
<td>255</td>
<td>0</td>
<td>255</td>
</tr>
</tbody>
</table>

specifies eight colors. As described in the section on color tables in the IDL manuals, these eight colors can be loaded into the first eight elements of the IDL default color table with the statements

\[
\text{IDL}\> \text{red} = [ 0, 0, 0, 255, 255, 255, 255 ] \\
\text{IDL}\> \text{green} = [ 0, 0, 255, 255, 0, 0, 255 ] \\
\text{IDL}\> \text{blue} = [ 0, 255, 0, 255, 0, 255, 0 ] \\
\text{IDL}\> \text{tvlct}, \text{red}, \text{green}, \text{blue}
\]

\(^{68}\)Setting the keyword `decomposed` to zero (false) is essential to guarantee that the `color` keyword in the `plot` command will refer to the selected color table.

\(^{69}\)RGB designations are the most straightforward. IDL also allows specification by HSV (hue-saturation-value) and HLS (Ostwald system), which are described in the IDL manuals.
where the command `tvlct` loads the specified color table into the proper internal storage. Then, in subsequent plot commands, the color red, for example, would be identified as color 4. Once these color values have been loaded into the current color table, the statements\(^\text{70}\)

IDL> x = findgen(101)/10.0
IDL> y = sin(x)
IDL> plot, x, y, yrange=[-2.0,10.0], /ndata ; Draw axes
IDL> for i = 0, 5 do oplot, x, y+i, thick=3.0, color=i+1

will draw a succession of six sine curves, one above the other and each with a different color.

If we want to write a color PostScript file so that the colors seen on the screen are reflected in the printed output (on a color printer), we must modify the process so far described for creating a PostScript file to invoke the keyword `color` to the `device` command (so that IDL’s current color table is incorporated into the file and, hence, downloaded to the printer). The statements

```idl
set_plot, 'ps'
device, filename='test.ps', /color
... execute plot commands
device, /close
set_plot, 'x'
```

will write the requisite file.

\textit{Once we are finished with our own private color table, we can use the statement `loadct,0` to return to the original default color table.}

\section*{2.17 Miscellaneous Occasionally Useful Tidbits}

In this section, we enumerate features of IDL that are used so infrequently that they are easily forgotten between uses.

\subsection*{2.17.1 Specifying Directories}

Different operating systems use different forms to specify the path that identifies a particular directory, for example, to set the default directory or to identify the location of a file to be read or written. The character separating directories in a path depends on the operating system. In UNIX, that character is a forward slash; in Windows, it is a backslash, though a single forward slash in an IDL path will be properly interpreted even in Windows.

\subsection*{2.17.2 IDL’s Default Directory}

If IDL’s CLI is launched from a shell window of the operating system, the default directory (current working directory) will be the directory from which IDL was launched. In the GUI, the default directory is displayed in the box labeled ‘Current Directory’ just above the panel in which commands are entered. In either interface, the default directory can be \textit{displayed} with the pair of statements

IDL> cd, current=direct & print, direct

\(\text{\footnotesize\textsuperscript{70}}\)In the initial plot command, we use the default color rather than color 7 for white so that, when we later produce hard copy (and the black background on the screen is turned to white), the axes (in white on the screen) will be turned to black in the hard copy. If we used color 7 (white), it would \textit{stay} white in the hard copy and the axes would be invisible against the (now) white background.
In either interface, the default directory can be changed with statements like

```
IDL> cd, 'FullPath'
```

Provide full path of new directory

```
IDL> cd, 'DirectoryName'
```

Shift focus to subdirectory of current directory

```
IDL> cd, '../DirectoryName'
```

.. backs up one directory

Alternatively, but only in the GUI, the default directory can be changed by editing the text in the box labeled ‘Current Directory’, which action will construct and execute an appropriate command that will be automatically displayed at the IDL> prompt.

When the default directory is unknown or ill-defined, including a full specification of a path in a quoted file name is necessary.

2.17.3 Customizing IDL

The configuration of IDL can be customized in two ways, specifically by

1. A startup command file, the path to which is provided in the environment variable `IDL_STARTUP` for the CLI and in the text entry box labeled ‘Startup file:’ in the Window/Preferences/IDL tab in the GUI and/or

2. A (separate) command file, which can be invoked at any time by a specific command submitted at the IDL prompt.

While individual users can invoke the first method, that route is more often reserved to the system manager, who can supply a standard startup file and thereby create a standard starting IDL environment for all users. The Local Guide will explain whether the end user needs to establish any environment variables for the CLI and/or enter a specific path in the Window/Preferences/IDL tab for the GUI). Most often, the proper parameters will be established without any action on the part of the end user.

The second method of customization is recommended for individual end users, who can use it by creating a supplementary startup command file whose execution after IDL is started will provide additional customization (and perhaps override some of the features established by the system-supplied startup file). Then, in the CLI, the statement

```
idl startup
```

to the operating system when IDL is started or, in either the CLI or the GUI, the IDL statement

```
@startup
```

issued as the first statement to IDL after IDL starts will execute the file `startup.pro` in the default directory. Whichever command is used, the system-wide startup file is executed first; then, the individual file will be executed. (The name of the individual startup file is, of course, arbitrary.)

2.17.4 IDL’s Search Path

Whenever an explicit path is not specified for a file that IDL must read (either data file or pro-file), the program looks first in the current default directory (see Section 2.17.2) and then in a sequence of directories defined by the search path, which is stored in the system variable `!path` and can at any time be displayed with the statement `print, !path`. The initial search path is defined by default or, when IDL is launched, by an optional statement in the file pointed to as described in item 1.
in Section 2.17.3. The specific initial search path may be site-specific and is identified in the Local Guide. A statement like\(^{71}\)

\[
!\text{path} = '/\text{usr}/\text{people/abcdef:} + !\text{path}
\]

will add the directory /usr/people/abcdef at the beginning of the current search path.

2.17.5 Restoring IDL’s Initial State

Sometimes, we would like to restore IDL to its initial state. The simplest way to achieve this objective is to exit from IDL and then restart it. IDL also provides the executive command .reset_session, which will remove all defined variables and functions (including common blocks) but will not reset system variables to their defaults. If you anticipate the need to restore those variables without exiting from IDL, you can save them immediately upon starting IDL with a statement like

\[
\text{save, } /\text{system\_variables, filename='defaultvars.dat'}
\]

(Choose any convenient file name.) Then, at any time later in the session (or in another session), you can restore the default values with the statement

\[
\text{restore, 'defaultvars.dat'}
\]

(The file name in the restore statement must, of course, match the one you used in the save statement, and a full path must be included in the filename if the current default directory does not contain the referenced file.)

2.17.6 Common Storage in IDL

Common storage was introduced in Section 2.8. There is, however, no way short of executing the executive command .reset_session or exiting from and then restarting IDL to redefine a common area of a given name once it has been created at IDL’s main command level. Since these routes both entail removing much more than the common area from IDL’s workspace, neither may provide in all circumstances an acceptable way to remove a common area. Note, however, that a common area defined at IDL’s main command level (i.e., at the IDL prompt IDL>) can contain more variables than are used in various procedures or functions subsequently called in the session. Thus, for example, if we know that we will need at most six variables in a common area of a particular name, we can define that area at the main command level with six variables. Subsequent routines referring to that named common can then use no more than six variables but can, in fact, use fewer, should fewer be sufficient for the purposes of that routine.

2.17.7 Reading Data from the Keyboard

Occasionally, and perhaps especially in various pro-files, we may wish to prompt for one or more values by displaying a message on the screen and then read those values from characters typed at the keyboard. The keyboard is identified to IDL as input channel 0, so that, when executed, the statement

\[
\text{readf, 0, x}
\]

\(^{71}\)Be aware that full paths to different directories are separated by a character that depends on the operating system (‘:\’ in UNIX and ‘;’ in Windows; see also the Local Guide). We use ‘:’ in this example.
will prompt with a colon ‘:’ and wait for the value of x to be entered from the keyboard; entry is terminated by pressing ⟨RETURN⟩. Alternatively, the statement

\texttt{read, x}

will accomplish the same end without the need to specify an input channel. Should a prompting message be desired, we need simply provide it in quotation marks (single or double) as a first argument, as for example in the statement

\texttt{read, 'Please enter a value for x: ', x}

or, alternatively, by using the keyword \texttt{prompt}, as in

\texttt{read, prompt = 'Please enter a value for x: ', x}

If two or more values are to be entered, then the statement requesting the entry would be

\texttt{read, 'Enter x and y coordinates: ', x, y}

and, separated by space(s) or a comma, the values will be entered in a single line.

\textbf{2.17.8 NaN and Inf}

Division by zero will produce a warning but also assign to the variable being evaluated the value \texttt{Inf}. Some other arithmetic infelicities will result in the assigning of the value \texttt{NaN}—not a number—to the variable being evaluated. Rather than terminating execution of a loop when one or the other of these events occurs, IDL simply assigns the appropriate value to the current variable and goes on to the next value of the loop index. Other IDL routines know how to ignore these special values. While ultimately we should develop procedures that do not encounter these errors, in quick operations and in the early development of a procedure, we will not be forced by IDL to straighten out these problems before meaningful results can be obtained in subsequent operations.

\textbf{2.18 References}

Those seeking additional information about IDL should

\begin{itemize}
  \item Consult the on-line help messages that can be brought up by typing ‘?’ at the IDL prompt or by accessing the \texttt{HELP} menu in the menubar of the GUI interface to IDL.
  \item Google 'IDL manuals' and examine some of the links that emerge, including in particular
    \begin{itemize}
      \item the link to IDL PDF guides,
        \url{www.harrisgeospatial.com/docs/PDF-GUIDES.html}
      \item the link to the Exelis Documentation Center
        \url{www.harrisgeospatial.com/docs/home.html}
    \end{itemize}
\end{itemize}

and
CHAPTER 2. INTRODUCTION TO IDL

- any of several links to documents produced by third-party entities.

You may also find in the library associated with your computational facilities one or more of the manuals

- *Learning IDL*, which provides a pedagogically oriented introduction to a large fraction of IDL’s capabilities and is an appropriate source for the first time user for whom the full reference manual may well seem daunting.

- *Using IDL*, which focuses on the interactive use of IDL and briefly describes the interface and many of the more commonly used commands for plotting functions, displaying images, processing signals, and performing various mathematical operations.

- *Building IDL Applications*, which focuses on the writing of programs in the IDL language.

- *The IDL Reference Guide*, which is a three-volume set detailing all capabilities of IDL and, in particular, containing a systematic, alphabetically arranged directory of all commands, option variables, keywords, and . . .

- *The IDL HandiGuide*, which—in ten or a dozen pages—summarizes all IDL commands, environment variables, system variables, graphics keywords, and other useful information.

While the versions available to you may be outdated (and some may no longer be in print), most of the information in any that you can find is almost certainly still accurate.

2.19 Exercises

2.19.1 Writing IDL Statements

2.1. Write and test IDL statements to create (a) a five-element column vector, (b) an 8 × 8 unit matrix, and (c) a 10 × 10 matrix all of whose elements are zero except those on the main diagonal (which are all 2) and those on the diagonals just above and just below the main diagonal (which are all −1). Search for a route more efficient than laboriously setting each of the 100 elements in the 10 × 10 matrix individually.

2.2. Look up the procedure *sort* both in the on-line help and in the printed IDL manuals. Then create a vector of your choice and test the use of *sort*, following the pattern illustrated in the documentation. Finally, write in your own words a brief description of what *sort* does.

2.3. Look up the procedures *lusol* and *ludc* both in the on-line help and in the printed IDL manuals, then use the process described therein to solve the linear equations solved by other means in Section 2.3.4.

2.4. Describe and test a sequence of IDL statements that uses a for/do loop to evaluate the dot product of two n-component vectors a and b. In essence, you will have to initialize a variable to zero and then, in the loop, successively add to that variable each of the products a[i]·b[i] in turn.

2.5. (a) Describe and test a sequence of IDL statements that uses a for/do loop to evaluate \( \sum a_i \) when the values of \( a_i \) are supplied as the elements of the vector a. In essence you will have to initialize a variable to zero and then, in the loop, successively add to that variable each of the elements \( a_i \) in turn. (b) Describe and test a sequence of IDL statements that uses the built-in function total to achieve the same end.

2.6. Write and test an IDL function that accepts two three-component vectors as input and returns a three-component vector containing the cross product of the two input vectors.
2.7. The Fibonacci numbers are generated by picking two starting values and then adding values successively, where each added value is generated by adding the previous two values, e.g., the list \([1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]\) shows the first several Fibonacci numbers when the first two are both 1. Write a program that asks for the first two numbers and the ultimate length of the desired list and then calculates and displays the corresponding Fibonacci sequence. See [math.temple.edu/~reich/Fib/fibo.html](http://math.temple.edu/~reich/Fib/fibo.html) for a fascinating discussion of the significance of Fibonacci numbers.

2.19.2 Finding Eigenvalues and Eigenvectors

2.8. Patterning your code after that in `lueigen`, write and test a function that accepts a symmetric matrix as input and returns a matrix, each column of which is structured with its first element one of the eigenvalues of the input matrix and its remaining elements the components of the eigenvector belonging to that eigenvalue.

2.9. Find the eigenvalues and eigenvectors for each of the following matrices:

(a) 

\[
\begin{pmatrix}
0 & 1 & 0 \\
1 & 0 & 1 \\
0 & 1 & 0
\end{pmatrix}
\]

(b) a 10 × 10 matrix with zeroes everywhere except that all elements on the main diagonal have the value 2.0 and all elements on the diagonals just above and just below the main diagonal have the value −1.0. Search for a route more efficient than laboriously setting each of the 100 elements in the 10 × 10 matrix individually.\(^\text{72}\)

(c) Identify the five eigenvalues in part (b) belonging to the lowest five eigenvalues and, for each, plot a graph whose vertical coordinate is the component of the eigenvector and whose horizontal coordinate is the component number, i.e., a graph of vec[i] versus i. Actually, in the underlying physical context, it would be more appropriate to plot graphs of the result of augmenting these eigenvectors by placing an element 0.0 both before the first element and after the last element in the eigenvector. If, for example, the eigenvectors are in the columns of `evecs`, then the statement

```idl
plot, [ [0.0], [ evecs[0,*] ], [0.0] ]
```

would plot the requested graph for the first column of `evecs`—though you should try to improve the appearance of the plot by tampering with the scales, adding labels, ... In particular, the statements

```idl
x = findgen(12)/11.0
plot, x, [ [0.0], [ evecs[0,*] ], [0.0] ]
```

will produce a graph whose horizontal axis is more suitably labeled. In such a display, you should see something close to the lowest several modes of a vibrating string fixed at both ends!

(d) \((\text{Optional})\) Repeat parts (b) and (c) but with a similarly constructed matrix that is 50 × 50.

(e) \((\text{Optional})\) Use procedure `sort` to sort the eigenvalues found in parts (b) and (d) into increasing order and then sort the matrix of eigenvectors to match.

2.10. When a (weak) constant external electric field of magnitude \(F\)—we reserve \(E\) for energy in this exercise—is imposed on a hydrogen atom, the energy of the states with principal quantum number

\[^{72}\text{Using the statement} \text{'print, matrix'} \text{ to print a } 10 \times 10 \text{ floating matrix } \text{matrix} \text{ will result in a printout that is difficult to read because each row of the matrix will occupy more than one line on the screen. If the matrix happens to contain only integers, you might use instead the statement 'print, fix(matrix)'. With this statement, each row of the matrix will take only one line on the screen. The statement 'print, fix(100*matrix)' will show more decimal places in a compact display, though you will need mentally to adjust all printed values by the extraneous factor of 100.\]
CHAPTER 2. INTRODUCTION TO IDL

2.19.3 Graphing Scalar Functions of a Single Variable

2.11. Create and test a sequence of IDL statements that will produce a graph that is similar to Fig. 2.6 except that the axes are drawn along the lines \(y = 0\) and \(x = 0\). Note the strategy used in Section 2.10.3.

2.12. (a) Create a vector of 201 elements whose values range from -10.0 to 10.0 in equal steps, (b) obtain a graph of the function

\[
y(x) = \frac{x}{a + b(x - c)^2}
\]

over the interval -10.0 \(\leq x \leq 10.0\) with \(a = 1.0\), \(b = 2.0\), and \(c = 1.0\), (c) explore the way the function depends on the three parameters \(a\), \(b\), and \(c\), and (d) write a paragraph describing that behavior in words.

2.13. Consider a circular disk of radius \(a\) lying in the \(xy\) plane with its center at the origin. If the disk carries a uniform charge on its surface, the electrostatic potential at the point \((0, 0, z)\) on the axis of the disk is given by

\[
V(z) = E_0 \left[ \sqrt{a^2 + z^2} - |z| \right]
\]

where \(E_0\) is a constant. Obtain a graph of \(V(z)/(E_0 a)\) versus \(z/a\). Hint: Remember the IDL function \(abs\).

2.14. The voltage drop across an initially uncharged capacitor in a series RC circuit that is connected at time \(t = 0\) to a battery is given by the expression

\[
V(t) = V_0 \left( 1 - e^{-t/RC} \right)
\]

Obtain a family of graphs showing \(V(t)/V_0\) versus \(t\) for various values of \(RC\), and write a paragraph describing these graphs.
2.15. In a Fabry-Perot interferometer, a very large number of waves, each out of phase with the previous one by an amount \( \delta \) and reduced in amplitude by a factor \( r \), \( 0 \leq r < 1 \), interfere. The resulting intensity is proportional to the expression

\[
I(\delta) = \frac{1}{1 - 2r \cos \delta + r^2}
\]

Obtain graphs of \( I(\delta) \) versus \( \delta \) over the interval \(-3\pi \leq \delta \leq 3\pi\) for various values of \( r \), and write a paragraph describing these graphs.

2.16. The thin lens equation

\[
\frac{1}{s} + \frac{1}{s'} = \frac{1}{f}
\]

relates object distance \( s \), image distance \( s' \), and focal length \( f \). Here, \( s > 0 \) for an object to the left of the lens, \( s' > 0 \) for an image to the right of the lens, and \( f > 0 \) for a converging lens. With \( s \) ranging from large positive to large negative values, obtain graphs of \( s' \) versus \( s \) for various values of \( f \) (both positive and negative), and write a paragraph describing these graphs.

2.17. According to the special theory of relativity, the mass \( m \), the momentum \( p \), and the kinetic energy \( K \) of a particle moving with speed \( v \) are given in terms of the rest mass \( m_0 \) and the speed of light \( c \) by the equations

\[
m = \frac{m_0}{\sqrt{1 - \beta^2}} ; \quad p = \frac{m_0 v}{\sqrt{1 - \beta^2}} ; \quad K = \frac{m_0 c^2}{\sqrt{1 - \beta^2}} - m_0 c^2
\]

where \( \beta = v/c \). Obtain graphs of \( m/m_0 \), \( p/m_0 c \), and \( K/m_0 c^2 \) versus \( \beta \), superimposing on each a graph of the corresponding non-relativistic expression, and write a paragraph describing these graphs.

2.18. In a vacuum, the transmission and reflection coefficients \( T \) and \( R \) of a dielectric film of thickness \( d \) and index of refraction \( n \) are given by the equations

\[
T = \frac{4n^2}{4n^2 + (n^2 - 1)^2 \sin^2(\kappa d)} \quad \text{and} \quad R = \frac{(n^2 - 1)^2 \sin^2(\kappa d)}{4n^2 + (n^2 - 1)^2 \sin^2(\kappa d)} \quad (2.22)
\]

where \( \kappa = 2\pi n/\lambda \) and \( \lambda \) is the wavelength of the wave in vacuum. Obtain graphs of \( T \) and \( R \) versus \( \lambda/d \) for various values of \( n \) and write a paragraph describing these graphs. Warning: Don’t try plotting too close to \( \lambda = 0 \) since the function \( \sin(\kappa d) \) gives trouble at that point.

2.19. Consider two circular disks, each of radius \( R \), located with their centers on the \( z \) axis such that their planes are parallel to the \( xy \) plane. Let the first disk have its center at the point \((0, 0, b/2)\) and the second at the point \((0, 0, -b/2)\) so that the disks are separated by a distance \( b \) (\( b > 0 \)) and the origin is halfway between them. If the top disk carries a uniform, constant charge density \( \sigma \) and the bottom disk carries a uniform, constant charge density \(-\sigma\), the electrostatic potential at the point \((0, 0, z)\) is given by

\[
V(z) = \frac{\sigma}{2\epsilon_0} \left[ \sqrt{R^2 + \left(\frac{b}{2}\right)^2} - \left| z - \frac{b}{2} \right| - \sqrt{R^2 + \left(\frac{b}{2}\right)^2} + \left| z + \frac{b}{2} \right| \right]
\]

Obtain graphs of \( V(z)/(\sigma R/2\epsilon_0) \) versus \( z/R \) for various values of \( b/R \) and write a paragraph describing these graphs. Hint: Remember the IDL function \texttt{abs}.

2.20. Using data from an experiment you have performed, create a suitable ASCII text file, read the data into IDL, and produce a graph showing the data, error bars, and a theoretical curve. Label the graph completely.

2.21. Journal articles often contain graphs showing data taken by several different scientists regarding the same relationship, with each experimentalist’s contribution marked with a different symbol. Describe an IDL procedure to produce such a graph and, if you have suitable data available (or if you can invent such data), demonstrate that your procedure works.
2.22. When a photon of initial energy $E_0$ undergoes Compton scattering from an atom of mass $m$ and is scattered by an angle $\theta$, the energy of the photon is reduced to

$$E(\theta) = \frac{E_0}{1 + \xi(1 + \cos \theta)}$$

where $\xi = E_0/mc^2$. Obtain both Cartesian and polar graphs of $E(\theta)/E_0$ versus $\theta$, $-\pi \leq \theta \leq \pi$, for several values of $\xi$, and write a paragraph describing these graphs.

2.23. A charged particle moves along the $z$ axis with speed $v$. When the particle passes through the origin, the magnitude of the electric field produced by the particle is given by the expression

$$E(\theta) = \frac{q}{4\pi \epsilon_0 r^2} \frac{1 - \beta^2}{(1 + \beta^2 \sin^2 \theta)^{3/2}}$$

where $\theta$ is the polar angle of the observation point, $r$ is the radial coordinate at that point, and $\beta = v/c$ ($c$ is the speed of light). Obtain graphs of $E(\theta)/(q/4\pi \epsilon_0 r^2)$ versus $\theta$ and, to reveal some of the behavior more visibly, of $E(\theta)/E(0)$ versus $\theta$ for various values of $\beta$ on the interval $-\pi \leq \theta \leq \pi$, and write a paragraph describing these graphs.

2.24. The intensity of the interference pattern produced by four slits illuminated by light of wavelength $\lambda$ when each slit is separated from the next by a distance $a$ is given by

$$I(\delta) = \cos^2 \delta (1 + \cos \delta)$$

where $\delta = 2\pi a \sin \theta/\lambda$. Obtain both Cartesian and polar graphs of $I$ versus $\theta$ on the interval $-\pi/2 \leq \theta \leq \pi/2$ for various values of $\lambda/a$, and write a paragraph describing these graphs.

2.19.4 Graphing Scalar Functions of Two Variables

2.25. The Planck radiation law gives the expression

$$u(\lambda, T) = \frac{8\pi ch}{\lambda^5} \frac{1}{e^{ch/(\lambda kT)} - 1}$$

for the distribution of energy in the radiation emitted by a black body. Here, $c$ is the speed of light, $h$ is Planck’s constant, $k$ is Boltzmann’s constant, $\lambda$ is the wavelength of the radiation, and $T$ is the absolute temperature. Using appropriate dimensionless units, plot this function (a) as a function of $\lambda$ for several $T$ and (b) as a surface over the $\lambda T$-plane. Write a paragraph about the way the peak changes in position, height, and width as $T$ changes. Hint: Choose a reference wavelength $\lambda_0$ arbitrarily and recast the expression in terms of the dimensionless variable $\lambda = \lambda/\lambda_0$. Then, note that $T_0 = ch/(\lambda_0 k)$ has the dimensions of temperature and re-express the temperature $T$ in terms of the dimensionless quantity $T = T/T_0$. (You might find it informative to evaluate $T_0$ for $\lambda_0 = 550$ nm.) With these changes, the expression to be plotted can be recast in the form

$$\pi(\lambda, T) = \frac{u(\lambda, T)}{8\pi ch/\lambda_0^5} = \frac{1/\lambda^5}{e^{1/(\lambda^5 T)} - 1}$$

and the question now becomes one of plotting this quantity using the dimensionless variables $\lambda$ and $T$.

2.26. A solenoid of length $L$ and circular cross-section of radius $a$ lies with its axis along the $z$ axis and its center at the origin. When the solenoid carries a current, the magnetic field at the point $(0,0,z)$ on the axis of the solenoid is given by

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

where $B_0$ is the magnetic field at the center when $a \ll L$, i.e., when the solenoid is effectively infinite in length. Plot graphs showing $B(z)/B_0$ (a) as a function of $z/L$ for various values of $a/L$, (b) as a surface over the $(z/L)(a/L)$ plane, and (c) as a contour over the $(z/L)(a/L)$ plane. Write a paragraph describing these graphs.
2.27. 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

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

where \( B_0 \) is the magnetic field at the origin. Plot graphs showing \( B(z)/B_0 \) (a) as a function of \( z/a \) for various values of \( b/a \), (b) as a surface over the \((z/a)(b/a)\) plane, and (c) as a contour over the \((z/a)(b/a)\) plane. Write a paragraph describing these graphs.

2.28. In an LRC circuit of resonant frequency \( \omega_0 \), the current \( I \) is given as a function of frequency \( \omega \) by

\[
I = \frac{I_0}{\sqrt{1 + Q^2 \left( \frac{\omega}{\omega_0} \right)^2}}
\]

where \( \omega = \omega/\omega_0 \). Plot \( I/I_0 \) (a) as a function of \( \omega \) for various values of the quality factor \( Q \) and (b) as a surface over the \( \omega Q \)-plane. Write a paragraph describing these graphs.

2.29. Refine the technique leading to Fig. 2.15 showing the displacement of a square drumhead so that, both on-screen and in hard copy, the display will actually be square. The approach described in Section 2.16.5 will be useful. \( \text{Hint:} \) Remember the keyword isotropic.

2.30. A spherical potato of radius \( a \) is taken from the refrigerator at 0 °C and placed in an oven at \( u_0 = 200 \) °C. The temperature \( u(r,t) \) at a point \( r \) from the center of the potato at time \( t \) is given by

\[
u(r,t)/u_0 = 1 - 2 \sum_{n=1}^{\infty} \frac{j_0(\beta_n r/a)}{\beta_n j_1(\beta_n)} e^{-\kappa \beta_n^2 t/(ca^2)}
\]

where \( \kappa \) is the thermal conductivity of the potato, \( c \) is its heat capacity per unit volume, \( j_0(x) \) and \( j_1(x) \) are the zeroth- and first-order spherical Bessel functions, and \( \beta_n \) is the \( n \)-th root of \( j_0(x) \), i.e., \( j_0(\beta_n) = 0 \). Obtain graphs of \( u(r,t)/u_0 \) as a function of \( r/a \) for various values of \( t \). Obtain also a graph of the temperature \( u(0,t) \) at the center of the potato as a function of \( t \) and determine how long it takes the potato to bake if, by being baked, one means that the temperature at the center has risen to 175 °C, i.e., to a value such that \( u(0,t)/u_0 = 0.875 \). \( \text{Hints:} \) (1) Note that

\[
j_0(x) = \frac{\sin x}{x} ; \quad j_1(x) = \frac{\sin x}{x^2} - \frac{\cos x}{x}
\]

Thus, the \( n \)-th root of \( j_0(x) \) is \( \beta_n = n \pi \). (2) Express times in units of \( ca^2/\kappa \) but then, taking the radius of the potato to be \( a = 0.05 \) m and taking \( \kappa \) and \( c \) for the potato to be those of water \( [\kappa = 0.63 \text{ J/(m s K)}, \text{ and } c = 4.2 \times 10^6 \text{ J/(K m}^3)] \), determine the unit in which your answers are expressed, both in seconds and in hours. (3) Experiment a bit, but note that the exponential factor decays more rapidly as \( n \) increases, so truncation of the infinite series at some point is probably in order.

2.31. At a particular time, a planet of mass \( M \) is located at the origin in the \( xy \) plane and a moon of mass \( M/3 \) is located at a point a distance \( R \) from the planet on the \( x \) axis. The gravitational potential energy of a spaceship of mass \( m \) at the point \((x,y,z)\) is then given by

\[
V(x,y,z) = -\frac{GmM}{\sqrt{x^2 + y^2 + z^2}} - \frac{GmM/3}{\sqrt{(x-R)^2 + y^2 + z^2}}
\]

Using IDL, obtain surface plots and contour maps of this potential energy in the \( xy \) plane (i.e., the plane \( z = 0 \)) and in the planes \( z = 0.1R \) and \( z = 0.5R \). \( \text{Suggestion:} \) Recast the function in dimensionless form by measuring \( x, y, \) and \( z \) in units of \( R \) and \( V(x,y,z) \) in units of \( GmM/R \).


2.19.5 Graphing Scalar Functions of Three Variables

2.32. Following the pattern illustrated in Section 2.13, explore at least one of the three-dimensional scalar fields

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

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

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

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

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 numerically.

2.33. 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

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

where \(l, m,\) and \(n\) are positive integers. Obtain several presentations of the pressure distribution inside this box at \(t = 0\) for several different values of \(l, m,\) and \(n\).

2.19.6 Graphing Vector Fields

2.34. Suppose the functions \(F_x(x, y)\) and \(F_y(x, y)\) give the \(x\) and \(y\) components of a vector field at the point \((x, y)\) in the \(xy\) plane. A very crude algorithm for determining the coordinates of points on the field line that starts at the point \((x_0, y_0)\) would entail the operations

\[ x_{old} \leftarrow x_0 \]
\[ y_{old} \leftarrow y_0 \]
\[ ds \leftarrow \text{chosen step size} \]

\[ \text{LOOP} \]

\[ \text{FX} \leftarrow F_x(x_{old}, y_{old}) \]
\[ \text{FY} \leftarrow F_y(x_{old}, y_{old}) \]
\[ \text{FM} \leftarrow \sqrt{\text{FX}^2 + \text{FY}^2} \]
\[ x_{new} \leftarrow x_{old} + ds*\text{FX}/\text{FM} \]
\[ y_{new} \leftarrow y_{old} + ds*\text{FY}/\text{FM} \]

\[ \text{Draw line from } (x_{old}, y_{old}) \text{ to } (x_{new}, y_{new}) \]
\[ x_{old} \leftarrow x_{new} \]
\[ y_{old} \leftarrow y_{new} \]

\[ \text{EXIT_LOOP} \text{ WHEN DONE} \]

\[ \text{END_LOOP} \]

Set starting point.
Choose step.
Start loop to generate steps.
Calculate components of force.
Calculate magnitude of force.
Step \(dx\) along force line.
Draw segment of force line.
Shift focus to new point.
Repeat.

Cast the loop in this algorithm as an IDL procedure \texttt{trace_field} that would be called with a statement like

\[ \texttt{trace_field, xold, yold, ds, n} \]

where \texttt{xold} and \texttt{yold} convey the desired starting point for a field line, \texttt{ds} conveys the desired step size, and \texttt{n} stipulates the number of steps to be taken before completing the line and returning to the calling program. Though we might ultimately want to provide a means to define the field
outside of the tracing procedure, for purposes of this exercise suppose that the field of interest is given by

\[
F_x(x, y) = \frac{x - 1.5}{[(x - 1.5)^2 + y^2]^{3/2}} - \frac{x + 1.5}{[(x + 1.5)^2 + y^2]^{3/2}}
\]

\[
F_y(x, y) = \frac{y}{[(x - 1.5)^2 + y^2]^{3/2}} - \frac{y}{[(x + 1.5)^2 + y^2]^{3/2}}
\]

which—in unspecified units—gives the electric field produced in the xy plane by two point charges, one of relative strength +1 located at \((x, y) = (1.5, 0)\) and the other of relative strength −1 located at \((x, y) = -(1.5, 0)\). As a first pass, suppose we are interested in the field only in the region \(-1 \leq x, y \leq 1\) but, once you have your program operating successfully, you can relax that constraint. (Be aware, however, in relaxing the constraint that your algorithm will probably give troubles if you try to plot too close to either of the point charges.) When executed, `trace_field` is to draw the field line that starts at the specified point and continues until \(n\) steps have been made. Structure your procedure so that its execution does not erase the current plot, i.e., so that you can execute it repeatedly with different starting points to build up a full map of the field of interest. Hint: You may want to use the command `plot` with the keyword `nodata` set to `true` to establish the axes before beginning to trace any field lines. Optional: Refine the procedure so that

(a) the loop is terminated not only after execution of a fixed number of steps but also when the field line has gone out of bounds, approaches too closely to a singular point, or returns to a point too close to its starting point (closed field line),

(b) it obtains its field definitions from two functions whose names are supplied as arguments when it is called, and/or

(c) it uses a more refined predictor-corrector algorithm in which the values `xnew` and `ynew` determined above are instead regarded as `xpred` and `ypred`, the field is calculated at that predicted point, and then a final step to `xnew` and `ynew` is made from `xold` and `yold` by using the average of the fields at the points \((xold, yold)\) and \((xpred, ypred)\).

2.35. Since equipotential curves are perpendicular to the field lines representing the associated (conservative) force field, an algorithm for tracing equipotential curves in two dimensions differs from the algorithm described in the previous exercise only by arranging for the stepping to occur at right angles to the calculated field rather than in the direction of the field. That objective is accomplished by replacing the two statements evaluating `xnew` and `ynew` in the algorithm in the previous exercise with the statements

\[
xnew \leftarrow xold - ds*FY/FM
\]

\[
ynew \leftarrow yold + ds*FX/FM
\]

Following a pattern similar to that described in the previous exercise, cast the loop in this algorithm as an IDL procedure `trace_equipot` that would be called with a statement like

`trace_equipot, xold, yold, ds, n`

where `xold` and `yold` convey the desired starting point for the equipotential curve, `ds` conveys the desired step size, and `n` stipulates the number of steps to be taken before completing the curve and returning to the calling program. Then, explore the equipotential curves depicting the field on which the previous exercise focussed.

2.36. Sometimes, interest can be focussed on the magnitude of the vector field rather than on the full field. A statement like

\[
MAG = \sqrt{Hx^2 + Hy^2}
\]

for example, will generate an array `MAG` containing the magnitudes of the magnetic field at each available grid point. This array then conveys a scalar field that can be displayed by any of the methods described in Section 2.12. Explore the magnetic field given in Eq. (2.19) in this way.
2.37. Read the IDL manuals about the procedures `flow3` and `scale3`. Then use `flow3` to explore the velocity field

\[ \mathbf{v}(x, y, z) = \omega(-y \mathbf{i} + x \mathbf{j}) + \alpha \mathbf{k} \]

(which happens not to depend on \( z \)) for various values of \( \omega \) and \( \alpha \). Then write a paragraph or two describing the field. Reassurance: Diagrams of vector fields in three dimensions are not easy to fathom. Do not be dismayed if the pictures you generate are initially mysterious. Hints: (1) Note that `flow3` will not function properly unless its invocation has been preceded by an appropriate invocation of `scale3`. (2) In establishing a scale, you may find it useful to assume a length scale \( a \), recast the field in the form

\[ \mathbf{v} = a\omega \left( -\frac{y}{a} \mathbf{i} + \frac{x}{a} \mathbf{j} \right) + \alpha \frac{\mathbf{k}}{a\omega} \]

and try to draw graphs of the field \( \mathbf{v}/a\omega \) as functions of the parameter \( \alpha/a\omega \) in the space whose axes are labeled \( x/a, y/a, \) and \( z/a \).

2.19.7 Animation

2.38. The transverse motion of a flexible string of length \( l \) lying nominally along the x axis and fixed at both ends can be expressed as the superposition

\[ y(x, t) = \sum_{n=1}^{\infty} A_n \sin \left( \frac{n\pi x}{l} \right) \cos \left( \frac{2\pi nt}{T} \right) \]

of its normal modes of oscillation. Here, \( A_n \) is the amplitude of the \( n \)-th harmonic (and may be negative to convey a 180° phase shift relative to a mode with positive amplitude) and \( T \) is the period of the fundamental mode of oscillation. In particular, the shape of the string at time \( t = 0 \) is given by

\[ y(x, 0) = \sum_{n=1}^{\infty} A_n \sin \left( \frac{n\pi x}{l} \right) \]

Measuring \( x \) in units of \( l \) and \( t \) in units of \( T \), generate animated displays by writing a procedure that will accept as input a vector giving the amplitudes of the first fifteen harmonics and produce a continuously running display showing the motion of the string when its initial shape is defined by those amplitudes. Test your program with a variety of sets of amplitudes, including but not limited to the first several harmonics by themselves. For example, when the string is pulled aside at its center and released from rest, the amplitude of the first several harmonics will be

1.0, 0.0, -0.111111, 0.0, 0.04, 0.0, -0.0204082, 0.0, 0.0123457, 0.0, 0.00826446, 0.0, 0.00591716, 0.0, -0.00444444

and, when it is pulled aside very near to one end, the amplitude of the \( n \)-th harmonic will be \( 1/n \).

2.39. The displacement of a string supporting a transverse wave is given by \( u = f(x, t) \) where \( f(x, t) \) is a given function of \( x \) and \( t \). Develop a general procedure to animate this wave propagation (i.e., for showing a sequence of images created by graphing \( f(x, t) \) as a function of \( x \) for a succession of values of \( t \)) and write an appropriate pro-file to accomplish this task. Try your procedure with the two functions

\[ f(x, t) = e^{-(x-\omega t)^2} \quad \text{and} \quad f(x, t) = \sin(x) \cos(t) \]

but feel free to invent others of your own choosing.

2.40. The displacement of a square membrane extending over the region \( 0 \leq x, y \leq a \) when it is oscillating in its \( m, n \) normal mode is given by

\[ u(x, y, t) = A \sin \left( \frac{m\pi x}{a} \right) \sin \left( \frac{n\pi y}{a} \right) \cos (\omega_{mn} t) \]

where \( \omega_{mn} = (c\pi/a) \sqrt{m^2 + n^2} \), \( c \) being the speed of propagation of the waves in the membrane. Write a procedure pro-file in IDL to animate the motion of this drumhead for a user-selected mode
(user-selected values of $m$ and $n$). *Suggestions:* (1) Express $x$ and $y$ in units of $a$ and $u$ in units of $A$ so the graphs you produce will show $u/A$ above the $(x/a)(y/a)$ plane. (2) Review the discussion in the last paragraph of Section 2.15.
2.A Listing of *lugen_grid*

```idl
pro lugen_grid, x, xrange=xrange, nx=nx, \$  
y, yrange=yrange, ny=ny, \$  
z, zrange=zrange, nz=nz  

; The procedure *lugen_grid* generates 2D and 3D arrays of evenly  
; spaced x, y, and z points. Information about the third  
; coordinate is optional and can be omitted, in which case two-  
; dimensional arrays will be produced.  
;  
; In the call to this procedure, x, y, z are arrays in which the  
; coordinates will be stored, xrange, yrange, zrange are two-component  
; vectors giving the range of values on each axis, and nx, ny, nz  
; are the number of subdivisions in each range.  
;-

; Set nz appropriately if not supplied  
if not keyword_set(nz) then nz = 0  

; Guard against entry of floating values  
xn=fix(nx) & yn=fix(ny) & zn=fix(nz)  

; Set spacing between grid points; create arrays for x and y  
dx = ( xrange[1] - xrange[0] )/float(nx)  
dy = ( yrange[1] - yrange[0] )/float(ny)  
x = fltarr( nx+1, ny+1, nz+1 )  
y = fltarr( nx+1, ny+1, nz+1 )  

; Set spacing and create array for z (if necessary)  
if nz ne 0 then begin  
   dz = ( zrange[1] - zrange[0] )/float(nz)  
z = fltarr( nx+1, ny+1, nz+1 )  
endif  

; Generate lists of values of x, y, and (if necessary) z  
xtmp = xrange[0] + dx*findgen(nx+1)  
ytmp = yrange[0] + dy*findgen(ny+1)  
if nz ne 0 then ztmp = zrange[0] + dz*findgen(nz+1)  

; Write values into arrays for x, y, and (if necessary) z  
for i = 0, nx do begin  
   for j = 0, ny do begin  
      for k = 0, nz do begin  
         x[i,j,k] = xtmp[i]  
y[i,j,k] = ytmp[j]  
      endfor  
   endfor  
endfor  
for i = 0, nx do begin  
   for j = 0, ny do begin  
      if nz ne 0 then z[i,j,k] = ztmp[k]  
   endfor  
endfor  
```

end
Chapter 7

Introduction to MAPLE

Note: All program (.mpl) files referred to in this chapter are available in the directory $HEAD/maple, where (as defined in the Local Guide) $HEAD must be replaced by the appropriate path for your site. At some sites, this directory or some other directory containing these files may also have been placed in MAPLE’s default search path. If so, the files can be found by MAPLE without explicit specification of a path. Otherwise, you will have to use the full path to copy them into your default directory to access them.

Originating with the Symbolic Computation Group at the University of Waterloo in Ontario, Canada, MAPLE® is a large program for manipulating expressions symbolically. MAPLE can perform symbolic algebra; evaluate derivatives, integrals, and Taylor series; solve algebraic and differential equations; manipulate complex numbers; find eigenvalues and eigenvectors; produce two- and three-dimensional graphs; and accomplish numerous other tasks. In this chapter, we describe some ways by which expressions can be defined for MAPLE and then illustrate some of the statements by which MAPLE can be instructed to manipulate these expressions. Users should also work through the on-line MAPLE “tour” (Section 7.2) and look at the examples with which most on-line help messages conclude (also Section 7.2). Further details can be found in assorted MAPLE documentation (Section 7.19) and in on-line help messages accessible from within MAPLE itself (Section 7.2 yet again). We shall refer to this documentation collectively as the MAPLE manuals.

Two user interfaces to MAPLE are provided by the vendor, a command-line interface (CLI), which runs directly in a Shell window of the operating system, and a graphical user interface (GUI), which displays equations in a more elegant form, offers assistance for those who have trouble remembering the commands, and has a notebook capability to facilitate documentation. With either interface, statements to MAPLE are structured and typed in the same way. Thus, in this chapter, we confine ourselves to describing the structure of statements without addressing the ways in which the GUI facilitates assembling them. We shall, however, present MAPLE output with the approximate appearance it will have in the GUI.

Except where occasionally noted otherwise, for the rest of this book, information provided describes the GUI explicitly and may not accurately reflect the behavior of the CLI.

1MAPLE is registered trademark of and also a commercially available program produced and marketed by Waterloo Maple, Inc. (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 Waterloo Maple, Inc. The terms of that license are explained in the Local Guide.
CHAPTER 7. INTRODUCTION TO MAPLE

7.1 Beginning a MAPLE Session

Detailed instructions for initiating a session with MAPLE will be found in the *Local Guide*. In general, MAPLE will be started either (1) by typing the command `maple` (for the CLI) or the command `xmaple` (for the GUI) at the prompt from the operating system,\(^2\) (2) by double-clicking ML on an appropriate icon on the desktop, or (3) by selecting ‘Command Line Maple n’ or ‘MAPLE n’ from a menu.\(^3\) We must from the outset be aware that

1. Internally, MAPLE is case sensitive, i.e., MAPLE treats upper and lower case letters as distinct. For the most part, built-in MAPLE commands use lower-case letters.

2. Lines of MAPLE code entered at its prompt can be terminated in any of three ways, depending on whether we want to
   
   (a) submit the statement for execution and instruct MAPLE to display its response, in which case we terminate the statement with a semicolon,
   
   (b) submit the statement for execution and instruct MAPLE to suppress explicit display of its response, in which case we terminate the statement with a colon, or
   
   (c) continue entering the statement on a new line, in which case we terminate the line with \(<\text{SHIFT}/\text{RETURN}>)\(^4\) and MAPLE will prompt for the next line in the CLI with the character > and in the GUI by simply advancing the cursor to a new (indented) line with no prompt.

   Whatever the terminating character, (RETURN) must always be typed for the interaction with MAPLE to continue.

3. Two or more statements can be placed in a single physical line by separating the statements with a semicolon (if output is to be produced) or a colon (if output is to be suppressed).

4. Typing (CONTROL/C) in the CLI or clicking ML on the ‘Stop’ button\(^5\) in the toolbar in the GUI will abort the current computation, display the MAPLE prompt, and await the entry of additional statements.

5. When using the CLI, the statement `quit;` or `quit:` (stop and done will also work) at any prompt for input will terminate execution, returning control to the computer’s operating system. In the GUI, exiting from MAPLE altogether is achieved by selecting ‘Exit’ from the File menu or clicking ML on the symbol \(\times\) in the upper right corner of the GUI; closing the current worksheet but keeping the GUI active is achieved by selecting ‘Close Document’ from the File menu. The statements `quit`, `done`, and `close` have been disabled in the most recent versions of MAPLE.\(^6\)

6. In the course of a lengthy MAPLE session, MAPLE’s workspace may become cluttered with things no longer needed. The MAPLE statement

   \(>\text{restart:}\)

\(^2\)In these commands to the operating system, case may be important.

\(^3\)Behind the scenes at the start of a MAPLE session, the start-up script checks to see that the license manager is running and that a license is available. If no license is available (or the license manager happens not to be running), the desired session will not start. Restarting a stopped license manager requires action by the system administrator.

\(^4\)On some keyboards, the (RETURN) key is labeled (ENTER).

\(^5\)The ‘Stop’ button is the icon with a white hand inscribed in a dark octagon. If you hover the cursor over the icon, the label ‘Interrupt the current operation’ will soon appear.

\(^6\)In some earlier versions of MAPLE, the statements `done`, `quit`, and `close` in the GUI will leave Maple running but close the current worksheet.
will return MAPLE to its state immediately after it was started in the first place (though—See Section 7.18—the command does not reinitialize interface parameters). In particular, initialization files—See Section 7.17.4—will be reread. Note, however, that—for complicated technical reasons—MAPLE may have less free memory available after a restart than it would have had we exited from MAPLE and restarted the program. Especially when preparing for a memory-intensive task, exiting and restarting the program from ground zero is preferable to invoking the command restart.

7. An initial exclamation point ‘!’ typed in the CLI allows temporary escape from MAPLE to the operating system. Thus, for example the statement \texttt{!lp file.ps} in UNIX issued to MAPLE in the CLI will be executed not by MAPLE but rather by the operating system, and any output will be displayed directly in the CLI. In the GUI on any platform, this feature is available (and will display any output in a pop-up window), but the feature is unnecessary, since we can easily use resources of the desktop to execute other programs without exiting from MAPLE. In either case, control is then returned to MAPLE.

If you start the CLI, the MAPLE prompt > will soon appear in the Shell window from which MAPLE was launched and you can proceed immediately to submit statements to MAPLE. Unless your local environment has been defined to be other than the standard default, starting the GUI interface for the first time in MAPLE10 or later will bring up an X-window in which you will be asked to select the ‘Document’ interface—the default if you make no explicit selection—or the ‘Worksheet’ interface. \textit{So that what you see on the screen comports most closely with what is described in this book, you should}

- Select the ‘Worksheet’ interface to create a blank worksheet within the GUI.\textsuperscript{7,8,9,10}
- Toggle the style in which statements typed to MAPLE will be displayed on the screen from the default ‘Maple Input’ to ‘Standard Math’ input by

  1. Opening the menu \texttt{Tools→Options} and clicking ML on the tab labeled ‘Display’,
  2. Toggling the selection labeled ‘Input display:’ from ‘2-D Math Notation’ to ‘Maple Notation’, and
  3. Clicking ML on the button ‘Apply Globally’ at the bottom of the \texttt{Options} window.

This change will not affect lines already displayed in your worksheet but all subsequent input lines in the current session will be displayed in the form to which you have become accustomed and, because you applied the change globally, this form for input lines will become the default in subsequent sessions as well.

\textsuperscript{7}Prior to MAPLE10, the command \texttt{xmaple} by default started what is now the ‘Worksheet’ interface; in MAPLE10 and thereafter, the command \texttt{xmaple} offers the options just described. Note, however, that these commands for starting MAPLE are frequently customized at individual sites. Please check your \texttt{Local Guide} for the commands applicable at your site.

\textsuperscript{8}In that worksheet, the MAPLE prompt > will be preceded by an opening square bracket \texttt{[>}. This difference has to do with the ability of the worksheet to record text and documentation. We will discuss this capability briefly in Section 7.16.

\textsuperscript{9}To create a new worksheet in the GUI, select ‘New’ from the \texttt{File} menu or click ML on the appropriate icon—probably the left most icon—in the toolbar.

\textsuperscript{10}Once you have made the choice by selecting one or the other interface with the mouse, that choice—and lots of other default parameters—will in the UNIX world be recorded in the file \texttt{/home/YourUserName/\ maple/VersionNumber/maplerc}. This “rc” file will control the interface you see in all subsequent sessions. (You can change many of the parameters, including your choice of ‘Worksheet’ or ‘Document’ interface, by selections available in the panels of the \texttt{Options} submenu of the \texttt{Tools} menu in the menu bar of the Worksheet interface.)
CHAPTER 7. INTRODUCTION TO MAPLE

7.2 On-Line Help

MAPLE makes several sorts of on-line help available. In the GUI, we can initiate a tutorial by selecting the tour from the HELP menu. The strategy for exiting from the tour while preserving the original MAPLE session may differ among the versions of MAPLE. In MAPLE16, the tour opens a second tab in the GUI interface. In that version, the tour can be terminated by selecting ‘Close Document’ from the File menu or by clicking ML on the symbol × at the right end of the tab associated with the tour. In either case, the original worksheet to MAPLE will return. Several other items, including ‘Maple Help’ and ‘Manuals, Resources, and more’ can be accessed from that same HELP menu in the GUI.

Beyond the tour, MAPLE contains an extensive on-line help library, which is organized into topics, subtopics, subsubtopics, .... In the GUI, simply typing a question mark ? followed by ⟨RETURN⟩ at the MAPLE prompt (or selecting ‘Maple Help’ from the HELP menu) brings up a browser in which we can navigate the hierarchy of the on-line help files. More explicitly, a statement like\footnote{Note that no terminating semicolon is required with the ? command.}

\(\textit{\texttt{?topic \ e.g., \ > \ ?index}}\)

will provide information about the specified entry, while the statement

\(\textit{\texttt{> \ ?index,packages}}\)

will list the packages about which information is available\footnote{Note that there is no space after the comma.} and the statement

\(\textit{\texttt{> \ ?inttrans}}\)

will list information about the package \texttt{inttrans} (integral transforms), whose presence would be revealed in the list produced by the statement \texttt{?index,packages}.\footnote{Because of the nature of a \textit{Shell} window, these statements have more limited action in the CLI.}

Items in the on-line help library that describe specific MAPLE commands typically include not only a description of the function and syntax of the command but also examples of the proper use of the command. Note particularly the on-line help messages for the commands \texttt{diff}, \texttt{int}, \texttt{simplify}, \texttt{expand}, \texttt{combine}, \texttt{factor}, \texttt{solve}, \texttt{sum}, \texttt{dsolve}, and \texttt{convert}.

7.3 Basic Entities in MAPLE

At the very beginning, we point out that MAPLE distinguishes among a variety of different entities, including

- \textit{expressions}, \textit{sequences}, \textit{sets}, \textit{lists}, \textit{arrays}, \textit{tables}, and \textit{character strings}, which are the primary quantities with which MAPLE works;
- \textit{commands} and \textit{functions}, which accept zero, one, or more arguments as input and return various outputs, thereby providing the means by which the objects enumerated in the previous item are manipulated;
- \textit{environment variables}, \textit{e.g.}, \texttt{Digits}, \texttt{Order}, and \texttt{Env*}, whose values—default or user-specified—modify the behavior of the associated functions;
7.4 Variable Names in MAPLE

Legal variable names begin with an alphabetic character or an underscore character and cannot coincide with reserved words like true, sin, and factor but, within reason, are otherwise unrestricted. They can contain alphabetic and numeric characters as well as the underscore character and can be of any length.

Lower case and upper case letters are treated as distinct. In the GUI, spelling out the name of a Greek letter in an input line will result in the display of the lower-case or upper-case Greek letter in the associated output line, e.g., delta in an input line will produce δ in the output line while Delta in an input line will produce Δ in the output line.

Note particularly that, in almost all situations, MAPLE will assume by default that any variable to which no explicit value has been assigned represents a complex quantity.

7.5 Expressions in MAPLE

Expressions are written in MAPLE using a syntax very much like that used in ordinary algebra and are supplied merely by typing the expression at MAPLE’s prompt for input, e.g.,

```maple
> x + a * cos(b*x);
```

to which, since the expression was terminated with a semicolon rather than a colon, MAPLE will respond

```
x + a cos(b x)
```

The symbols +, -, *, /, and ^ are used for addition, subtraction, multiplication, division, and exponentiation (though, paralleling FORTRAN, MAPLE also recognizes ** for exponentiation). Except for those in quoted strings, spaces are ignored and may be freely inserted for legibility. Functions available for use in expressions include

- General functions, e.g., abs(), evalf(), trunc(), round(), sqrt(), signum(), and rand().
- Trigonometric, hyperbolic and exponential functions and their inverses, e.g., sin(), cos(), tan(), arcsin(), arccos(), arctan(), sinh(), cosh(), tanh(), arcsinh(), arccosh(), arctanh(), exp(), and ln().

---

14 The underscore character, however, should be avoided as an initial character because MAPLE uses variable names beginning with underscores internally, and the risk of conflict is not zero.

15 Since almost all MAPLE commands and special words use entirely lower case letters, using upper-case letters in user-defined variable names reduces the likelihood of conflict. We will, however, only rarely follow this advice.

16 The actual maximum length depends on the operating system, but is quite large even in the most restrictive case.

17 Note that we will occasionally rearrange the display of MAPLE output lines to improve legibility. No substance is lost in that rearrangement.
• Special functions, e.g., Bessel and modified Bessel functions, Airy functions, elliptic integrals, gamma and beta functions, the error function, Legendre polynomials, Hermite polynomials, and Laguerre polynomials.

Full details on these and other functions, including information about the form of their arguments, will be found in the MAPLE manuals. A few will be introduced in the remainder of this chapter.

Internally, MAPLE sees simple expressions as lists. The first element (part—or operand—zero of the expression) indicates the operator involved, and the remaining elements (known as part—or operand—one, part—or operand—two, ...) indicate the operands. Thus, for example, the two expressions

\[ a + b + c + d \quad \text{and} \quad abc \]

are stored internally as the lists

\[ [ +, a, b, c, d ] \quad \text{and} \quad [ *, a, b, c ] \]

respectively. More complicated expressions, such as

\[ a + b - c + d \quad \text{and} \quad a + b(c + d) + e \]

are stored as lists of lists, being

\[ [ +, a, b, [-, c], d ] \quad \text{and} \quad [ *, a, [*, b, [+, c, d ] ], e ] \]

respectively. As a final example, which anticipates a later need, suppose the expression of interest is

\[ (x + 3)^2(x - a)(x + a) \]

We would generate MAPLE’s internal representation in several steps finding

\[ [ *, (x+3)^2, x-a, x+a ] \]

\[ [ *, [\^, x+3, 2], [+, x, -a], [+, x, a] ] \]

\[ [ *, [\^, [+, x, 3], 2], [+, x, [-, a]], [+, x, a] ] \]

Here, the first line shows parts 0, 1, 2, and 3 of the original expression while the second line reveals that part 1 of the original expression is itself a list having part 0 (part 0 of part 1 of the original expression; \(^\)\), part 1 (part 1 of part 1; \(x+3\)), and part 2 (part 2 of part 1; 2). Further, the third line reveals that some of the parts of the parts themselves have parts. Thus, for example, part 2 of part 1 of part 1 of the original expression is 3 while part 1 of part 2 of part 2 of the original expression is \(a\). These relationships can be involved and confusing, but some manipulations depend critically on being able to identify the parts, subparts, subsubparts, and ... of a complicated expression.

### 7.6 Assigning Values to Variables; Defining Functions

Especially when elaborate expressions are involved, we may want to refer to these expressions after they have been entered without having to retype them. If, for example, we anticipate frequent reference to a particular expression, we may wish to use the assignment operator ‘:=’ to assign or bind the expression to a variable. The statement
7.6. Assigning Values to Variables; Defining Functions

> bigexp := x + a * cos(b*x);
bigexp := x + a cos(bx)

for example, associates the variable bigexp with the expression following the operator :=. Subsequently, whenever that variable is included in an expression, it will automatically be replaced by the expression to which it has been bound, e.g., had we bound the expression to the variable bigexp, we might later have written

> bigexp^2;
(x + a cos(bx))^2

The wisdom of choosing variable names with mnemonic significance should be obvious.

Once a name has been assigned, we can at any time remind ourselves of the value bound to that variable simply by asserting its name. The statement

> bigexp;
x + a cos(bx)

for example, will request MAPLE to display the current value of bigexp. Further, if we can’t remember whether we have already assigned a value to a particular variable, we can invoke the statement

> assigned( bigexp );
true

to find out. Finally, if we have no further need for a particular variable and wish to tell MAPLE to forget that we had once bound a value to it, we could enter the statement

> bigexp := 'bigexp':

to remove the variable and free some of MAPLE’s workspace. In essence, this statement assigns the value of the variable to itself.

When special functions of significance to a particular session would be useful, we might exploit the MAPLE operators := and -> to append one or more user-defined functions to the built-in functions. For example, the statements

> spectrig := x -> 1 + sin(x)^2;
spectrig := x → 1 + sin(x)^2
> mode := (x,y) -> sin(Pi*x)*sin(2*Pi*y);
mode := (x, y) → sin(πx) sin(2πy)

define the functions spectrig(x) and mode(x,y) for subsequent use just as we would use any of MAPLE’s built-in functions. Sometimes, we might find it convenient to regard a statement defining a function alternatively as defining a transformation rule, which in effect converts whatever appears to the left of the symbol -> into whatever appears on the right. Further, if we have no further need for a particular function and wish to tell MAPLE to forget that we had once associated a function with a particular variable, we could enter the statements

> spectrig := 'spectrig':
> mode := 'mode':

18 The statement will, of course, return false if no value has been assigned to the specified variable.
to remove the functions and free some of MAPLE’s workspace.

Please note that a statement of the form \( p(t) := t^2 + \sin(3a\cdot t) \) does \textit{not} define a function; it merely introduces the symbol \( p(t) \) for the quantity on the right of the operator \( := \). With this definition, reference to \( p(t) \) will appear to do what you expect it to do, but the notation \( p(2\cdot x) \), for example, \textit{will not} return \( 4x^2 + \sin(6ax) \). Try to imprint this distinction firmly and quickly into your understanding.

Within Maple, some potential variable names are \textit{protected}. The names \texttt{Pi} (for \( \pi \)), \texttt{gamma} (for Euler’s constant, 0.577216), and \texttt{sin} (for the sine function), for example, are protected. An attempt to assign a value to a protected variable will fail, as illustrated in the statement

\[
> \text{gamma} := 3.0;
\]

\texttt{Error, attempting to assign to ‘gamma’ which is protected}

In rare circumstances, we may wish to use the symbol anyway. If so, we must \texttt{unprotect} it before using it in the new context, as illustrated in the statements

\[
> \text{unprotect( gamma ): gamma := 25.0:}
\]

Once we are finished with the non-standard use, we can restore the original status of the variable with the statements

\[
> \text{gamma := ‘gamma’: protect( gamma ):}
\]

Tampering with specially defined variables and other reserved words, however, is risky, and should be undertaken only occasionally and only for good reason.

### 7.7 Creating and Examining Basic Entities

We have already mentioned in Section 7.3 that MAPLE works with several different types of object and we have in Sections 7.5 and 7.6 described how to create expressions and assign values to variables. To simplify the quick tour of selected capabilities to be presented in Section 7.8, we include here a description of how some of the more important objects in MAPLE can be created and examined. \textit{Then}, we will be ready to introduce features that allow the processing of these various objects.

#### 7.7.1 Sequences

In MAPLE, a \textit{sequence} is an ordered, comma-separated “list” of items that is either \textit{not} enclosed in any delimiters or enclosed in parentheses.\(^{19}\) Sequences—data type \texttt{exprseq}—are created with statements like

\[
> \text{sq1 := a, b, a, d, e;}
\]

\[
\text{sq1} := a, b, a, d, e
\]

\[
> \text{sq2 := ( "David", 5.00, a+b );}
\]

\[
\text{sq2} := "David", 5.00, a + b
\]

Note that elements in a sequence may be duplicated and that the various items in the sequence need not have the same data type, though normally they will. The entire sequence can be referred to by its name, while a particular element in the sequence can be referred to by placing its index in square brackets after the name of the sequence, e.g.

\(^{19}\)In this section, we will place the word \textit{list} in quotation marks when the common meaning is intended. Without quotation marks, the word will mean a specific one of MAPLE’s internal objects.
7.7. CREATING AND EXAMINING BASIC ENTITIES

> sq2;
   "David", 5.00, a + b
> sq2[2];
   5.00

(Note that indices start at 1.)

7.7.2 Sets

In MAPLE, a set is a comma-separated “list” of elements that is enclosed in braces, \{\ldots\}, as delimiters. Sets—data type set—are created with statements like:\footnote{MAPLE is not entirely consistent across platforms and between the CLI and the GUI in the ordering of the terms or factors in its response to particular statements. In displaying MAPLE’s responses in this book, we will favor the display produced in the GUI, though the GUIs on different platforms do not always produce the same ordering. It even appears as if the same expression entered in different contexts on the same platform with the same interface doesn’t always yield the same ordering in the displayed output. Even though the details of the display vary, the different displays are, of course, always mathematically equivalent.}

> st1 := \{a, b, a, d, e\};
   st1 := \{a, b, d, e\}
> st2 := \{"David", 5.00, a+b\};
   st2 := \{ "David", a + b, 5.00 \}

With sets, only the presence or absence of a particular element is of concern. Hence, duplicate items in an assignment will be removed and order is not necessarily preserved. As with sequences, an entire set can be referred to by its name, while a particular element in the set can be referred to by placing its index in square brackets after the name of the set. Since the order of elements in a set is not important, however, we cannot always be sure which element will be in which position as a set is manipulated.

7.7.3 Lists

In contrast, a list is an ordered, comma-separated “list” of elements that is enclosed in square brackets, [\ldots], as delimiters. Lists—data type list—are created with statements like

> lst1 := [a, b, a, d, e];
   lst1 := [a, b, a, d, e]
> lst2 := ["David", 5.00, a+b];
   lst2 := [ "David", 5.00, a + b]

As with sequences and sets, an entire list can be referred to by its name, while a particular element in the list can be referred to by placing its index in square brackets after the name of the list. In contrast to sets, duplicate items in a list will be retained and order will be preserved.

7.7.4 Vectors, Arrays, and Matrices

MAPLE provides several ways to create one, two, and higher dimensional aggregates of elements, all of which have the same data type. One can, for example, create a vector by first dimensioning the array (so MAPLE knows how much storage to set aside) and then assigning values to the elements. We might, for example submit the statements
> Vec1 := Vector( 1..5 );

\[
\begin{bmatrix}
0 \\
0 \\
0 \\
0 \\
0
\end{bmatrix}
\]

Dimension vector with indices running from 1 to 5, inclusive. Note that this statement assigns the value zero to each element and creates a column vector, data type Vector\_column.

> for i from 1 to 5 do Vec1[i] := lst1[i]; end do:

Assign more specific values to elements; suppressing output.

> Vec1[2];

Display second element.

> Vec1;

Display entire vector.

(Here, \(\text{lst1}\) is the list created in the previous section.) To create a row vector instead of a column vector with this command, specify the optional argument as in \text{Vector(orientation=row, 1..5)}, data type Vector\_row. Note that, even to display a vector, the entire vector of either type can be referred to by its name while a particular element of the vector can be referred to by placing the corresponding number in square brackets after the name.\(^{21}\)

The command \text{Vector} can create only vectors. The command \text{Array} is more versatile. It can create a one-dimensional row array with statements like

> Arr1 := Array( 1..5 );

\[
[0 0 0 0 0]
\]

Dimension “vector” with indices running from 1 to 5, inclusive. Note that this statement assigns the value zero to each element. Note also that the “vector” created has data type Array.

> for i from 1 to 5 do vec2[i] := lst1[i]; end do:

Assign values to elements.

> Arr1[2];

Display second element.

> Arr1;

Display entire vector.

The command \text{Array} can also, however, create arrays of higher dimensions.\(^{22}\) To create a two-dimensional array, for example, we might exploit slightly more elaborate statements like

> Arr2 := Array(1..2, 1..2):

Dimension matrix, suppressing output. Note that this statement will create a 2 x 2 array of zeroes with data type Array.

> Arr2[1,1]:=1: Arr2[1,2]:=2:

Set elements.

> Arr2[2,1]:=3: Arr2[2,2]:=4:

Display array.

\[
\begin{bmatrix}
1 & 2 \\
3 & 4
\end{bmatrix}
\]

In this context, note that—in accordance with the usual conventions—the first index in a matrix

\(^{21}\)Some older versions of MAPLE would display the entire vector only if the display was requested with a statement like print(Vec1). Newer versions of MAPLE are not as restrictive.

\(^{22}\)Note that the earlier MAPLE command array has been “deprecated”, i.e., made obsolete—though for backward compatibility, MAPLE still responds to that command.
is the row index and the second index is the column index. Note also that, while we have started indices at 1, they can in fact be started at any (positive or negative) integer or even at zero.

Finally, matrices can be created by invoking the MAPLE command `Matrix`. For example, to create a two-dimensional matrix, we might invoke the statements

```maple
> Mat1 := Matrix(1..2, 1..2):
> Mat1[1,1]:=1: Mat1[1,2]:=2:
> Mat1[2,1]:=3: Mat1[2,2]:=4:
> Mat1;

\[
\begin{bmatrix}
1 & 2 \\
3 & 4
\end{bmatrix}
\]
```

Dimension matrix, suppressing output. Note that this statement will create a \(2 \times 2\) array of zeroes with data type `Matrix`.

Set elements.

Display matrix.

Because these structures are among the most frequently needed MAPLE objects, simpler methods for creating them are available. For example, the commands `Vector`, `Array` and `Matrix` admit an additional argument that specifies the actual elements in the array to be created. For a vector, that argument is simply a list of the elements, e.g.,

```maple
> Vec2 := Vector( 1..5, lst1 );

Vec2 := \[
\begin{bmatrix}
a & b & a & d & e
\end{bmatrix}
\]
```

For an array or a matrix, that argument is a list of the rows in the array or matrix, where each row is conveyed by a list of its elements.

```maple
> Arr3 := Array( 1..2, 1..2, [ [1,2],[3,4] ] );

Arr3 := \[
\begin{bmatrix}
1 & 2 \\
3 & 4
\end{bmatrix}
\]
```

```maple
> Mat2 := Matrix( 1..2, 1..2, [ [1,2],[3,4] ] );

Mat2 := \[
\begin{bmatrix}
1 & 2 \\
3 & 4
\end{bmatrix}
\]
```

Indeed, these commands are even capable of figuring out for themselves what the dimensions should be, so the statements

```maple
> Vec3 := Vector( lst1 );
> Arr4 := Array( [ [1,2],[3,4] ] );
> Mat3 := Matrix( [ [1,2],[3,4] ] );
```

will also create the corresponding objects in MAPLE’s workspace.

An even simpler way to create these structures is to exploit angle brackets. Elements in a row are delimited with vertical bars; those in a column are delimited with commas, e.g.,

```maple
> Vec4 := < a | b | c >;

Vec4 := [a b c]
```
> Vec5 := < a, b, c >;
Vec5 := \[
\begin{bmatrix}
a \\
b \\
c
\end{bmatrix}
\]
Mat4 := < a, b | c, d | e, f >;
Mat4 := \[
\begin{bmatrix}
a & c & e \\
b & d & f
\end{bmatrix}
\]
where these objects are of data types Vector\text{\textsubscript{row}}, Vector\text{\textsubscript{column}}, and Matrix, respectively.

The above discussion barely scratches the surface of the capabilities available with a wide assortment of options possible in the commands \texttt{Vector}, \texttt{Array}, and \texttt{Matrix}, and you are urged to examine the on-line help pages for these commands. Be aware, however, that vectors, arrays, and matrices involve different internal data structures, not all of which are meaningful in all contexts and that it may occasionally be necessary to convert from one to the other in the midst of a calculation. (See Section 7.7.5.)

The supplementary package \texttt{LinearAlgebra}, which we will introduce in Section 7.9, contains additional commands for creating and manipulating vectors and matrices.\textsuperscript{23}

\section*{7.7.5 Conversions}

Sometimes, the same entities appearing in different contexts will be needed in different forms. Thus, MAPLE provides the command \texttt{convert}, which has numerous features for transforming entities in one form to other compatible forms. For example, the conversion of a list to a set and a vector might be accomplished with the statements

\begin{verbatim}
> convert( lst1, set );
\{a, b, d, e\}
> convert( lst1, Array );
[a, b, a, d, e]
\end{verbatim}

respectively, while the conversion of a matrix to a list of lists could be accomplished with the statement

\begin{verbatim}
> convert( Mat1, listlist );
[[1, 2], [3, 4]]
\end{verbatim}

and the conversion of an array to a matrix could be accomplished with the statement

\begin{verbatim}
> convert( Arr1, Matrix );
[[1, 2], [3, 4]]
\end{verbatim}

We will see additional capabilities of this command as this chapter unfolds. The full list of keywords that will be accepted as the second argument to \texttt{convert} is included in the on-line help message displayed with the statement \texttt{?convert}.

\textsuperscript{23}While the older package \texttt{linalg} remains available, it has been declared obsolete and is deprecated, which means it may disappear from some later version of MAPLE. Commands in that package still work but they should be avoided.
7.8 Using MAPLE...

At base, a session with MAPLE involves entering one or more expressions and then instructing MAPLE to process these expressions in some way. The following transcripts of MAPLE sessions illustrate some of the simpler manipulations. The transcripts are presented in two columns, the one on the left recording the statements submitted to MAPLE and MAPLE's responses, the one on the right containing explanatory comments.

Note that the manner of editing statements submitted to MAPLE depends on whether the statement is entered in a CLI or in the GUI. With UNIX (which for the past several years includes the Macintosh operating system), MAPLE's CLI is a standard X-window interface, so

- the arrow keys can be used in the usual way to retrieve previous statements and move the cursor within a statement,
- the backspace and delete keys can be used to remove characters,
- other keys can be used to insert characters at the position of the cursor,
- anything already existing on the screen can be highlighted and then copied to the current input point by clicking MR, and
- regardless of the position of the cursor in a line, the entire statement will be submitted when \( \langle \text{RETURN} \rangle \). is typed.

Further, all of these editing features except the capacity to copy and paste are also available in the Windows CLI.

In the GUI, in-line editing can be accomplished in the same way using the left and right arrow keys to move the cursor, the backspace or delete key to remove characters, and other keys to insert new characters. The up and down arrow keys, however, move the cursor not only through the statements on input lines but also through components of the displayed output. The cursor can be located in previous statements by typing the up and down arrow keys a sufficient number of times (or by repositioning the cursor with a click of ML), the statement can then be edited, and the edited form reexecuted, but the new statement will take the place of the old; it will not be copied to the new input focus in the window. Alternatively, again in the GUI, a previous statement (or portion thereof) can be highlighted, copied into the clipboard by selecting ‘Copy’ from the Edit menu, and then—after clicking ML at the desired point of insertion—selecting ‘Paste’ from the Edit menu. Note that, whatever interface is in use, the cursor need not be positioned at the end of a line when the line is to be submitted for execution by pressing \( \langle \text{RETURN} \rangle \).: The full line will be submitted regardless of the position of the cursor within the line.

To keep the several segments in this section somewhat separated from one another, we shall periodically exit and then reenter MAPLE, thereby clearing MAPLE's workspace. The same end could be accomplished with the statement \texttt{restart};, as described in item 6 in Section 7.1 (though see the caution in that section). The action is, of course, not necessary, and the entire “conversation” could have been held perfectly well without these occasional breaks or—for that matter—with breaks in different places, so long as the breaks do not occur between the point at which a variable is defined and the point at which it is used. The “conversation” in this section should be started in a fresh invocation of MAPLE.

Finally, note that, throughout this section, various components of MAPLE (functions, commands, operators, packages, ... ) are introduced simply by presenting statements in which they are used, frequently with brief comment. Details on all mentioned components and on many others can be found in the MAPLE manuals.\footnote{On many keyboards, the \( \langle \text{RETURN} \rangle \) key is labeled \( \langle \text{ENTER} \rangle \).} \footnote{Input statements will look much the same regardless of the platform on which we are operating. Output is here presented \textit{approximately} as it would appear in the GUI. In the CLI, the content of the output will be the same but, because that interface is limited to character output, its appearance will be less “pretty”.}
7.8.1 ... for Arithmetic

We illustrate first some of the ways in which MAPLE can be instructed to perform simple arithmetic operations. Consider, for example, the statements

\[ 2 + 3; \]
\[ 5 \]

Enter a simple numeric expression. MAPLE returns a value, *doing* the arithmetic.

\[ 4/6; \]
\[ 2 \]
\[ \frac{2}{3} \]

Enter another expression. MAPLE does not automatically generate a floating point evaluation but *does* reduce the fraction.

\[ \sqrt{9}; \]
\[ 3 \]

Compute \( \sqrt{9} \).

\[ \sqrt{12}; \]
\[ 2 \sqrt{3} \]

Compute \( \sqrt{12} \). MAPLE simplifies the expression but keeps it exact.

\[ \text{factorial}(9); \]
\[ 362880 \]

Find \( 9! = 9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 \), with *factorial*.

\[ 9!; \]
\[ 362880 \]

Find 9 factorial with postfix operator !.

\[ \text{evalf}(4/6); \]
\[ 0.6666666667 \]

Ask for floating value with default number of digits.

\[ \text{evalf}(4/6, 20); \]
\[ 0.66666666666666666667 \]

Ask for floating value with 20 digits.

The number of digits displayed in a floating evaluation is determined by the environment variable *Digits*,\(^{26}\) whose default value is 10. To find values to different numbers of digits, we might use the statements

\[ \text{Digits}; \]
\[ 10 \]

Display default precision of floating evaluations.

\[ \text{Digits} := 32:\] \]

Change precision to 32 digits. Here, the colon at the end of the statement suppresses display but not behind-the-scenes evaluation of MAPLE’s response.

\[ \text{evalf}(\Pi); \]
\[ 3.1415926535897932384626433832795 \]

Ask for 32-digit floating value of \( \pi \), represented in MAPLE with the constant \( \Pi \).

\[ \text{Digits} := 10:\] \]

Reset default value of *Digits*. Here, we terminated the statement with a colon to suppress output.

Restart MAPLE. (See item 6 in Section 7.1.)

We have, of course, illustrated only some (+, -, sqrt, factorial, !, evalf) of the operators and functions that might be used to persuade MAPLE to manipulate with numbers and only one (*Digits*) of the environment variables whose values affect the behavior of these functions. You should make it a point to read about these and other functions, environment variables, constants, and operators in the MAPLE manuals.

7.8.2 ... for Algebra

MAPLE is, of course, capable of much more than simple arithmetic. We next illustrate how algebraic manipulations can be requested, beginning with the statements

\(^{26}\)Note the *upper-case D*. 
> (x+3)^2*(x^2-a^2);

$(x + 3)^2(x^2 - a^2)$

Enter an expression. Note that—depending on platform and interface and perhaps prior context—the order of the factors in the output may differ from that in the input. Compare the response produced on your system by this statement with the response produced by the statement $(x^2-a^2)*(x+3)^2$.

> expand(%);

$x^4 - x^2a^2 + 6x^3 - 6xa^2 + 9x^2 - 9a^2$

Expand the expression. Again, the order of terms may be platform, interface, and context dependent. Here, the symbol % refers to the most immediate past output. ($%\%\% \text{refers to the penultimate and ante-penultimate outputs, respectively.}$)

> fct := factor(%);

\[ fct := -(x + 3)^2(a - x)(a + x) \]

Factor the expression in the previous output.

We note, of course, that this factoring hasn’t quite returned us to our starting point. To achieve that end, we need to multiply the second and third factors without expanding the first. To do so, we will have to carve the expression apart, manipulate with its parts, and then reconstruct the expression. As discussed in Section 7.5, MAPLE sees this expression as a list of lists of lists… Indeed, we can persuade MAPLE to reveal the first level in that structure by exploiting the MAPLE function \texttt{op} (for operand) in the statement\textsuperscript{27}.

> [op(0,fct), op(1,fct), op(2,fct), op(3,fct), op(4, fct)];

\[ [\ast, -1, (x + 3)^2, a - x, a + x] \]

or we could exploit not only the function \texttt{op} but also MAPLE’s ability to construct a loop in the statement

> for i from 0 to 4 do op(i,fct); end do;

\begin{align*}
&-1 \\
&(x + 3)^2 \\
&a - x \\
&a + x
\end{align*}

to show all operands of the expression at once. Even more simply, we could use the command \texttt{op} with but one argument, finding that

> op( fct );

\[-1, (x + 3)^2, a - x, a + x\]

i.e., finding the operands of the expression as a sequence, though part 0 (which gives the operator affecting the other operands) is not revealed. If we really want to know part 0, we must ask for it explicitly, either with the statement

> op( 0, fct );

\begin{align*}
&\ast \\
&-1, (x + 3)^2, a - x, a + x
\end{align*}

\textsuperscript{27}The order of operands may be platform, interface, and perhaps context dependent. Even the number of operands may vary. For example, on some platforms, the result above might not have an initial minus sign and have $(x-a)(x+a)$ as the last two factors. Compare the parts identified here by the integers 0, 1, 2, 3, and 4, with the parts identified by those integers in your environment, where part 4 may be absent.
or, probably better, by invoking another MAPLE command in the statement

\[
> \text{whattype( fct );}
\]

Related to the function \text{op}, MAPLE supplies also the function \text{nops}, which returns the number of operands in the expression given as its argument, i.e.,

\[
> \text{nops( fct );}
\]

Note that, here also, part 0 is not counted.

Using the capabilities of the command \text{op}, we might achieve the entire objective of recasting the expression \text{fct} with the single nested statement

\[
> \text{fct1 := op(2,fct) * expand(op(1,fct)*op(3,fct)*op(4,fct) );}
\]

\[
fct1 := (x + 3)^2(x^2 - a^2)
\]

in which we extract parts 1, 2, and 3 from \text{fct}, expand the product of parts 2 and 3, and then reconstruct the expression by multiplying that result by part 1. This time, we have succeeded in reconstructing the original expression (except possibly for the order of the factors).\(^{28}\) As here illustrated, beating MAPLE’s version of a complicated expression into a more conventional form may require a fair bit of creative massaging.

We continue with some additional manipulations, first setting the constant \(a\) to the specific value 2 with the statement

\[
> \text{eval( fct1, a=2 );}
\]

\[
(x + 3)^2(x^2 - 4)
\]

Evaluate expression with \(a = 2\).

Then, for example, we could find a partial fraction expansion of the reciprocal of this polynomial with the statement

\[
> \text{eq1 := convert( 1/%, parfrac, x );}
\]

\[
\text{eq1} := \frac{1}{100(x - 2)} + \frac{1}{5(x + 3)^2} - \frac{1}{4(x + 2)} + \frac{6}{25(x + 3)}
\]

Find partial fraction expansion with respect to \(x\).

Alternatively, we could replace \(x\) with \(3z\) by using the \text{subs} command in the statement

\[
> \text{subs( x = 3*z, eq1 );}
\]

\[
\text{eq1} := \frac{1}{100(3z - 2)} + \frac{1}{5(3z + 3)^2} - \frac{1}{4(3z + 2)} + \frac{6}{25(3z + 3)}
\]

Substitute \(3z\) for \(x\) in the previous expression.

To illustrate a more complicated recasting, suppose that we wanted to square the denominator of the second\(^{29}\) term in \text{eq1} without affecting any other term. The statement \text{expand(eq1)} does nothing. (Try it.) We need a more sophisticated operation that rearranges only the second term (part 2 of the original expression). To achieve the desired end, we extract that term, carve it up, manipulate with the proper piece, and substitute that piece into the proper point in the original expression.

\(^{28}\)We repeat the substance of the previous footnote: The order and number of the operands in \text{fct} may be platform, interface, and perhaps context dependent. Thus, the numbers identifying various operands in your environment may differ from those used here, and your end result may have an initial minus sign and a sign difference in one of the factors. \text{Beware!}

\(^{29}\)Because the order of terms may vary from platform to platform, the desired term in your display may not be the second term.
expression. The task takes several steps, beginning with the identification of the proper part with the statements

\[ \text{eq2 := \text{denom( op( 2, eq1 ) )}}; \]

\[ eq2 := 5(x + 3)^2 \]

\[ \text{eq3 := \text{expand(op(2,eq2));} } \]

\[ eq3 := x^2 + 6x + 9 \]

\[ \text{expand( op( 2, denom(op(2,eq1)) ) )}; \]

\[ x^2 + 6x + 9 \]

Now, note that

\[ \text{op( 2, op( 2, eq1 ) )}; \]

\[ \frac{1}{(x + 3)^2} \]

Thus, we can attain the second term of the desired end by substituting the reciprocal of eq3 for part 2 of part 2 in eq1 with the statement

\[ \text{eq4 := subsop(2=1/eq3, op(2, eq1) );} \]

\[ eq4 := \frac{1}{5(x^2 + 6x + 9)} \]

and then substitute that result for part 2 in eq1 with the statement

\[ \text{eq5 := subsop( 2=eq4, eq1 );} \]

\[ eq5 := \frac{1}{100(x - 2)} + \frac{1}{5(x^2 + 6x + 9)} - \frac{1}{4(x + 2)} + \frac{6}{25(x + 3)} \]

Finally, we recognize the frequent need to bring together terms in an expression when those terms happen to have a common factor even if some terms in the expression don’t have that factor and the command factor will therefore not accomplish the desired task. The command collect is built for that task. Thus, for example, we might want to effect the recasting

\[ A \cos \omega t + B e^{-\gamma t} \cos \omega t + F \rightarrow (A + Be^{-\gamma t}) \cos \omega t + F \]

You can readily discover that the command factor will not do the job. Instead, the objective could be accomplished with the statements

\[ \text{eqn := A*cos(omega*t) + B*exp(-gamma*t)*cos(omega*t) + F;} \]

\[ eqn := A\cos(\omega t) + Be^{-\gamma t} \cos(\omega t) + F \]

\[ \text{collect( eqn, cos(omega*t) );} \]

\[ \cos(\omega t)(A + Be^{-\gamma t}) + F \]

Restart MAPLE. (See item 6 in Section 7.1.)

We have, of course, only mentioned a few (expand, factor, op, nops, eval, convert, subs, subsop, collect) of the numerous functions MAPLE makes available for manipulating expressions algebraically. We have also introduced the control structure for do/end do and the special argument parfrac to the command convert. The differences among several commands, including expand, simplify, combine, and factor are explored in one of the exercises. You should make it a point to read about these functions and option variables in the MAPLE manuals.
7.8.3 ... for Complex Variables

MAPLE is also capable of manipulating complex variables. Indeed, in most—though not all—contexts, MAPLE assumes that any variable to which no specific value has been assigned represents a complex value. We illustrate some of MAPLE’s main capabilities in this category with the statements

\[ fct := \sin(x+y*I); \]
\[ fct := \sin(x + Iy) \]
\[ \text{evalc}(fct); \]
\[ \sin x \cosh y + I \cos x \sinh y \]
\[ \text{evalc}(\text{Re}(fct)); \]
\[ \sin x \cosh y \]
\[ \text{evalc}(\text{Im}(fct)); \]
\[ \cos x \sinh y \]
\[ \text{evalc}(\text{conjugate}(fct)); \]
\[ \sin x \cosh y - I \cos x \sinh y \]
\[ fct1 := (x+I*y)/(x-I*y); \]
\[ fct1 := \frac{x + Iy}{x - Iy} \]
\[ \text{evalc}(\text{abs}(fct1)); \]
\[ \sqrt{\frac{x^2}{x^2+y^2} - \frac{y^2}{x^2+y^2}} + 4 \frac{y^2 x^2}{(x^2+y^2)^2} \]
\[ \text{evalc}(\text{argument}(fct1)); \]
\[ \arctan\left(\frac{2 y x}{x^2 - y^2}, \frac{x^2 + y^2}{x^2 - y^2}\right) \]
\[ \text{evalc}(r*exp(I*theta)); \]
\[ r \cos \theta + I r \sin \theta \]

Restart MAPLE. (See item 6 in Section 7.1.)

\[ r e^{I\theta} \]
\[ \text{evalc}(\%); \]
\[ r \cos \theta + I r \sin \theta \]

\[ \text{evalc}(\%); \]
\[ 1 \]

\[ \text{evalc}(\text{argument}(fct1)); \]
\[ \arctan(2 y x, x^2 - y^2) \]
\[ \text{evalc}(\text{abs}(fct1)); \]
\[ \sqrt{\frac{x^2}{x^2+y^2} - \frac{y^2}{x^2+y^2}} + 4 \frac{y^2 x^2}{(x^2+y^2)^2} \]

\[ \text{evalc}(\text{argument}(fct1)); \]
\[ \arctan(2 y x, x^2 - y^2) \]
\[ \text{evalc}(\text{abs}(fct1)); \]
\[ r e^{I\theta} \]

In some versions of MAPLE (or in some configurations), the command \texttt{polar} may not be available until the statement \texttt{readlib( polar )} has been executed.
7.8. USING MAPLE ...

We have, of course, only mentioned some (evalc, Re, Im, conjugate, abs, argument, polar) of the functions MAPLE makes available for manipulating complex numbers. Significantly, we have also introduced the command map for processing each part of an expression with the same function, and we have seen additional contexts in which the command simplify can be valuable. You should make it a point to read about these functions and any associated special arguments in the MAPLE manuals.

7.8.4 ... for Trigonometry

MAPLE also makes available a number of commands and functions for manipulating with trigonometric, hyperbolic, and exponential functions. Note, for example, the capabilities illustrated in the statements

```maple
> trig1 := sin(x)*cos(x);
  trig1 := cos x sin x
> convert( trig1, exp );
  \[ \frac{1}{2} i \left( e^{ix} - e^{-ix} \right) \left( \frac{1}{2} e^{ix} + \frac{1}{2} e^{-ix} \right) \]
> simplify( evalc( convert(%,trig) ) );
  \sin x \cos x
> combine( trig1 );
  \sin(2x)
> expand( % );
  \sin x \cos x
> trig2 := sin(2*x+y);
  trig2 := \sin(2x+y)
> expand( trig2 );
  2 \cos y \sin x \cos x + 2 \sin y \cos^2 x - \sin y
> expand( trig2, 2*x );
  \sin(2x) \cos y + \cos(2x) \sin y
> expand( % );
  2 \cos y \sin x \cos x + 2 \sin y \cos^2 x - \sin y
> (cos(x)+sin(x))^2;
  (\sin x + \cos x)^2
> expand( % );
  \cos^2 x + 2 \cos x \sin x + \sin^2 x
> simplify( % );
  2 \sin x \cos x + 1
> combine( % );
  \sin(2x) + 1
```

To illustrate the use of a loop once again, let us create a list of the values of the sine of integer multiples of \( \pi/2 \). We use the statements

```maple
> a := Array( 1..10 ):
> for i from 1 to 10 do a[i] := i-1; end do:
> a;
  [0 1 2 3 4 5 6 7 8 9]
> map( x -> sin(Pi*x/2), a );
  [0 1 0 -1 0 1 0 -1 0 1]
```

Restart MAPLE. (See item 6 in Section 7.1.)
We have, of course, only mentioned a few (\texttt{convert}, \texttt{expand}, \texttt{combine}, \texttt{evalc}, and \texttt{simplify}) of the functions MAPLE makes available for doing trigonometry. Additionally, we have seen two more second arguments for \texttt{convert} (\texttt{exp} and \texttt{trig}), and we have seen additional contexts in which the commands \texttt{Array} and \texttt{map} are useful. You should make it a point to read about these commands and any special arguments in the MAPLE manuals.

7.8.5 ... for Algebraic Equations

We turn next to illustrating a few MAPLE commands for manipulating expressions algebraically. To define a polynomial and find its roots, for example, we would use statements like

\begin{verbatim}
> poly := expand( (x^2-a^2) * (x+3)^2 );
> soln := solve( {poly = 0}, {x} );
soln := \{x = -a\}, \{x = a\}, \{x = -3\}, \{x = -3\}
\end{verbatim}

Define a polynomial.

Solve the equation \( poly = 0 \) for \( x \). There are four roots, though two of them are identical. We could alternatively submit the statement \( \texttt{solve(\{poly\},\{x\})} \); in which case MAPLE would have understood the equation to be \( poly = 0 \) and returned the same output.

Note that the arguments to the command \texttt{solve} are \texttt{sets}, and that the roots have been returned as a \texttt{sequence} of sets—a behavior that would seem more reasonable had we been solving a system of equations for more than one unknown. In the simple case of a single equation and a single unknown, the braces can be omitted, as in

\begin{verbatim}
> soln1 := solve( poly=0, x );
soln1 := -a, a, -3, -3
\end{verbatim}

With this form, MAPLE returns the roots as a simple sequence.

Of course, once we have obtained the roots, we should verify their correctness by backsubstitution into the original equation. To do so, we would execute statements like

\begin{verbatim}
> eval( poly, x = -a );
0
> for rt from 1 to 4 do
   eval( poly, soln[rt] );
end do;
0
0
0
0
\end{verbatim}

Verify first root.

Verify all roots, using the \texttt{for do/end do} structure to code a loop. Alternatively, we could have used the statement

\begin{verbatim}
for rt in soln do
   eval( poly, rt );
end do;
\end{verbatim}

Perhaps the most famous of all solutions is that of a quadratic equation. The command \texttt{solve} clearly knows the quadratic formula, as illustrated by the statements

\begin{verbatim}
> a*x^2 + b*x + c:
> solve( %, {x} );
\begin{align*}
\{ x &= \frac{-b + \sqrt{b^2 - 4ac}}{2a} \\
\{ x &= \frac{-b - \sqrt{b^2 - 4ac}}{2a} \}
\end{align*}
> eval( %, \{a=3, b=4, c=2\} );
\begin{align*}
\{ x &= -\frac{2}{3} + \frac{1}{6} \sqrt{-8} \\
\{ x &= -\frac{2}{3} - \frac{1}{6} \sqrt{-8} \}
\end{align*}
\end{verbatim}

Supply a quadratic expression.

Solve \( ax^2 + bx + c = 0 \) for \( x \). Note again that the roots have been returned as a sequence of sets.

Evaluate solution with indicated values of \( a, b, \) and \( c \).
We can, of course, always ask for floating point evaluations with the command \texttt{evalf}, finding here that

\begin{verbatim}
> evalf( % );
\{x = -.6666666667 + .4714052091I, x = -.6666666667 - .4714052091I\}
\end{verbatim}

We have accepted the default value (10) of \texttt{Digits}, which controls the number of digits in the floating point values returned by \texttt{evalf}.

Finally, let us illustrate MAPLE’s capability to solve simultaneous linear equations. To define and solve a representative system, we invoke the statements

\begin{verbatim}
> eqn1 := x - y = 4;  \quad \text{Give MAPLE a pair of linear equations, binding each to a variable to facilitate subsequent reference.}
  eqn1 := x - y = 4
> eqn2 := x + y = 0;
  eqn2 := x + y = 0
> solve( {eqn1,eqn2}, {x,y} ); \quad \text{Solve the system for } x \text{ and } y.
  \{y = -2, x = 2\}
\end{verbatim}

Note yet again that each solution—here there is only one—is presented as a set.

\begin{verbatim}
> eqn1 := a*x+b*y = e; eqn2 := c*x+d*y = f; \quad \text{Try it with a more symbolic system.}
> eqn1 := a*x+b*y = e; eqn2 := c*x+d*y = f;
> solve( {eqn1,eqn2}, {x,y} );
  \{x = \frac{-bf - ed}{ad - cb}, y = \frac{af - ce}{ad - cb}\}
\end{verbatim}

Remove knowledge of \texttt{eqn1}, \texttt{eqn2} from MAPLE’s memory.

\begin{verbatim}
\end{verbatim}

We have, of course, only mentioned one (\texttt{solve}) of the functions MAPLE makes available for solving algebraic equations exactly. Your attention is drawn also to the function \texttt{fsolve} for finding solutions numerically and to the function \texttt{LinearSolve}, which is built to solve matrix equations of the form $Ax = b$ and is contained in the supplementary package \texttt{LinearAlgebra}—a MAPLE feature more fully described in Section 7.8.6. You should make it a point to read about these functions in the MAPLE manuals, giving particular attention to the few environment variables whose values influence the behavior of the functions. Rootfinding is discussed in greater depth in Chapter 14.

\section*{7.8.6 \ldots for Manipulating Arrays}

Vectors, arrays, and matrices represent numerous physical quantities. We have already seen in Section 7.7 how to create these objects. Once they have been created, MAPLE can manipulate these objects in the standard ways. Most of those capabilities, however, are not provided in MAPLE’s primary set of commands. Instead, they are provided in a separate package called the \texttt{LinearAlgebra} package. There are several ways to invoke these supplementary capabilities. We can anticipate use of one or more commands in the full package by adding the full package to the MAPLE environment with the statement\footnote{terminated with a semicolon, this statement will display a list of all the commands contained in the package. To suppress that list, terminate the statement with a colon.}

\begin{verbatim}
with( LinearAlgebra );
\end{verbatim}
or we can anticipate use of a single command—say Transpose—in the package by adding that command alone to the MAPLE environment with the statement

```
with( LinearAlgebra, Transpose );
```

which identifies both the package (first argument) and the specific command (second argument) we want;\(^{32}\) either way, the command is then available to us with the name Transpose, e.g.,

```
tran := Transpose( M );
```

Alternatively (and without explicitly loading the entire package or some subset of its commands), we can invoke a particular command within the package by using its full name, e.g.,

```
tran := LinearAlgebra[Transpose]( M );
```

Packages will be more fully discussed in Section 7.9.

With this preamble, we are ready to illustrate some of MAPLE’s capabilities for manipulating these objects. We begin by loading the package LinearAlgebra, defining a matrix, and evaluating its transpose with the statements

```
> with( LinearAlgebra );
```

Load package LinearAlgebra.

```
> Mat1 := Matrix( 1..2, 1..2, [[1,2],[3,4]] );
```

Create Mat1.

```
Mat1 :=
[ 1 2 ]
[ 3 4 ]
```

```
> Mat2 := Transpose( mat1 );
```

Evaluate transpose of Mat1.

```
Mat2 :=
[ 1 3 ]
[ 2 4 ]
```

Then, we evaluate the inverse, the trace, and the determinant of the matrix with the statements

```
> MatrixInverse( Mat1 );
```

Evaluate inverse of Mat1. Note that the MatrixInverse command requires input of data type Matrix; input of type Array will not work.

```
[ -2 1 ]
[ 3 -1 ]
[ 2 -2 ]
```

```
> val1 := Trace( Mat1 );
```

Evaluate trace (sum of diagonal elements) of Mat1. Note that the Trace command will also work with objects of data type Array.

```
val1 := 5
```

```
> val2 := Determinant( Mat1 );
```

Evaluate determinant of Mat1. Note that the Determinant command will also work with objects of data type Array.

```
val2 := -2
```

Other common manipulations include evaluating the standard matrix product and an element-by-element product. The standard matrix product can be evaluated with the command Multiply in the package LinearAlgebra using, for example, a statement like

```
> Multiply( Mat1, Mat2 );
```

Evaluate the result of ordinary matrix multiplication.

```
[ 5 11 ]
[ 11 25 ]
```

Element-by-element multiplication is a bit trickier. One might think that the operator \( \ast \) for commutative multiplication could be used in a statement like

\(^{32}\) Additional commands in the package can be included as third, fourth, \ldots arguments.
7.8. USING MAPLE …

> Mat1*Mat2;

Try simple multiplication.

but this statement generates an error message. The problem is that commutative multiplication of matrices—i.e., objects of data type Matrix—is not defined. Instead, the “matrices” to be multiplied must be stored with the data type Array. Thus, to achieve an element-by-element multiplication of two matrices, we use the succession of statements

> Arr1 := convert( Mat1, Array );

Create first array.

> Arr2 := convert( Mat2, Array );

Create second array.

> ArrProd := Arr1*Arr2;

Evaluate the result of element by element multiplication.

Since Arr2 is also the transpose of Arr1, one might think that Arr2 could be created with the statement Transpose(Arr1). Unfortunately, the output of the Transpose command has data type Matrix, not Array, even when the input to that command has data type Array. Paying careful attention to data types in this area is both difficult and important.

Finally, MAPLE—more specifically, MAPLE’s package LinearAlgebra—includes functions for finding eigenvalues and eigenvectors, which we illustrate with the statements

> a := 1/sqrt(2):
> spinx := Matrix( [[0,a,0], [a, 0,a], [0, a, 0] ] );

Enter a matrix bound to the variable spinx.

(Multiline statements are achieved by typing the carriage return without a ‘;’ or a ‘:’.)

spinx :=

\[
\begin{bmatrix}
0 & \frac{1}{2} & \sqrt{2} & 0 \\
\frac{1}{2} & \sqrt{2} & 0 & \frac{1}{2} \\
0 & \frac{1}{2} & \sqrt{2} & 0
\end{bmatrix}
\]

> (e,v) := Eigenvectors( spinx );

Find eigenvectors and eigenvalues. Note that the eigenvectors may be returned in your context with a different normalization than those shown here.

Since there are three eigenvalues, the output of Eigenvectors consists of a vector e—data type VectorColumn—of the eigenvalues and a matrix—data type Matrix—in which the n-th column contains the eigenvector corresponding to the n-th eigenvalue. Note that the eigenvectors as returned by Eigenvectors are not normalized; the function Normalize (which requires input having data type Vector) in the package LinearAlgebra returns a normalized vector when given an unnormalized one. For example we can extract the first eigenvector from v and normalize it with the statements

> Normalize( Column(v,1), Euclidean );

Restart MAPLE. (See item 6 in Section 7.1.)
We have, of course, only mentioned a few (Transpose, MatrixInverse, Trace, Determinant, Eigenvectors, Normalize, Column) of the functions available in the package LinearAlgebra for manipulating matrices. Important additional functions in this package include Add, CrossProduct, DeleteColumn, DeleteRow, DotProduct, Eigenvalues, IdentityMatrix, LinearSolve, MatrixAdd, MatrixMatrixMultiply, MatrixScalarMultiply, Row, SubMatrix, VectorAdd, VectorMatrixMultiply, VectorScalarMultiply, ZeroMatrix, and ZeroVector. You should make it a point to read about these functions and any associated arguments in the MAPLE manuals.

7.8.7 ... for Basic Calculus

Differentiation, indefinite and definite integration, and determination of series representations of symbolic expressions are also among MAPLE’s capabilities. To evaluate derivatives and integrals, for example, we would use statements like

\[ fct := x^2 \cdot e^{-x^2}; \]
\[ \text{diff( fct, x );} \]
\[ \text{int( } fct, x ) ; \]
\[ \text{simplify( } \% , x ) ; \]
\[ \text{assume( a, positive );} \]
\[ \text{int( } x/(a^2+x^2)^{3/2} , x=0..\infty ) ; \]

Finally, (for this section), we point out that MAPLE can generate series expansions of prescribed functions. To illustrate, we ask MAPLE to process the statements

\[ \text{33 The command assume is extremely versatile. We shall return to it in Section 7.18. Note right now, however, that any variable that is the object of an assumption will be flagged by MAPLE with a suffixed tilde, as shown in this result. Adding assumptions to a variable already so identified must be done with the command additionally, since using assume again will replace previous assumptions with the new ones. The command about(VariableName) will provide a report revealing current assumptions. These commands and a means to suppress the flagging of variables that are affected by an assumption are more fully discussed in Section 7.18.} \]
> 1/(1+exp(-x));
\[
\frac{1}{1 + e^{-x}}
\]
Enter yet another expression.

> ser := taylor( %, x=0, 7 );
\[
ser := \frac{1}{2} + \frac{1}{4} x - \frac{1}{48} x^3 + \frac{1}{480} x^5 + O(x^7)
\]
Expand the expression in a Taylor series in \( x \) about \( x = 0 \), retaining terms up to (but not including) those of 7-th order in \( x \). Note here that the coefficients of the terms in \( x^2, x^4, \) and \( x^6 \) are all zero. (The default—controlled by the environment variable \texttt{Order}—is 6-th order.)

Occasionally, we may need to find a power series expansion in more than one variable. Paralleling the command \texttt{taylor}, MAPLE also provides the command \texttt{mtaylor} for finding multivariate Taylor series. For example, the statement

> mtaylor( 1/sqrt(1+x+y), \{x=2,y=0\}, 3 );
\[
\frac{1}{3\sqrt{3}} - \frac{1}{18\sqrt{3}} y - \frac{1}{18\sqrt{3}} (x - 2) + \frac{1}{72\sqrt{3}} y^2 + \frac{1}{36\sqrt{3}} (x - 2) y + \frac{1}{72\sqrt{3}} (x - 2)^2
\]
generates a two-dimensional Taylor series about the point \( x = 2, y = 0 \), expanding to include terms through those second-order in the variables.

One warning about series is in order. While a series displays on the screen as an algebraic expression, internally its form differs from that of many other expressions. The statements

> whattype( ser );
\texttt{series}

> op( ser );
\[
\frac{1}{2}, 0, \frac{1}{4}, 1, -\frac{1}{48}, \frac{1}{3}, \frac{1}{480}, 5, O(1), 7
\]
reveal that the expression type of a series is \texttt{series} and that the series itself is stored as a sequence of \texttt{pairs} of values. The first member of each pair is the coefficient of a particular term in the series and the second member of each pair is the power of the variable in that term. That variable itself is stored as part 0 of the series, as revealed by the statement

> op( 0, ser );
\( x \)

For some operations, it may be necessary to convert the series into a polynomial with the statement

> pol := convert( ser, polynom );
\[
\texttt{pol} := \frac{1}{2} + \frac{1}{4} x - \frac{1}{48} x^3 + \frac{1}{480} x^5
\]

34In some versions of MAPLE (or in some configurations), the command \texttt{mtaylor} may not be available until the statement \texttt{readlib( mtaylor )} has been executed.
Note that the omitted terms in the series are *not* indicated in \texttt{pol} and that \texttt{pol} has expression type \texttt{+}. Further, compare the difference between the *series* storage as revealed by the statement \texttt{op(ser)} and the *polynomial* storage as revealed by the statement \texttt{op(pol)}.

Beyond differentiation, integration, and evaluation of series, MAPLE is able to evaluate limits with statements like

\begin{verbatim}
> limit( sin(b*x)/x, x = 0 );
\end{verbatim}

\begin{verbatim}
> limit( (3*x^2+b^2)/(x^2-c^2), x = infinity );
3
\end{verbatim}

Restart MAPLE. (See item 6 in Section 7.1.)

The commands for evaluating derivatives, integrals, and limits (and a few other commands) actually have two different forms. In the above illustrations we have used only the forms \texttt{diff}, \texttt{int}, and \texttt{limit}. Sometimes, we may want instead to use the forms \texttt{Diff}, \texttt{Int}, and \texttt{Limit}. As we have seen in the above, the commands beginning with a *lower*-case letter will result in immediate evaluation of any derivatives or integrals that *can* be evaluated. In contrast, the commands beginning with an *upper*-case letter—called in Maple jargon the *inert* forms of the commands—will suppress evaluation of derivatives until a later time when that evaluation is explicitly requested. To illustrate the difference, suppose we define the function

\begin{verbatim}
> f := y -> a * sin( omega * y );
\end{verbatim}

\begin{verbatim}
f := y → a \text{sin}(\omega y)
\end{verbatim}

Then, the two statements

\begin{verbatim}
> diff( f(y), y );
\end{verbatim}

\begin{verbatim}
a \text{cos}(\omega y) \omega
\end{verbatim}

\begin{verbatim}
> d1 := Diff( f(y), y );
\end{verbatim}

\begin{verbatim}
d1 := \frac{\partial}{\partial y} (a \text{sin}(\omega y))
\end{verbatim}

reveal the difference.\textsuperscript{35} In the first, the derivative is evaluated; in the second, it is left unevaluated. At some later time, we could evaluate the derivative in this last expression with the statement

\begin{verbatim}
> value( d1 );
\end{verbatim}

\begin{verbatim}
a \text{cos}(\omega y) \omega
\end{verbatim}

The behavior of \texttt{int} and \texttt{Int} and of \texttt{limit} and \texttt{Limit} parallels that described for \texttt{diff} and \texttt{Diff}.

We have, of course, only mentioned a few (\texttt{diff}, \texttt{Diff}, \texttt{int}, \texttt{Int}, \texttt{taylor}, \texttt{limit}, \texttt{Limit}) of the numerous functions MAPLE makes available for doing calculus. Important additional functions include \texttt{series} and \texttt{asympt}. Further, additional functions and embellishments of some of those functions we have mentioned are components in the packages \texttt{student}, \texttt{DEtools}, and \texttt{PDEtools}. You should make it a point to read about these functions and packages in the MAPLE manuals.

\textsuperscript{35}Some versions of MAPLE use \texttt{d} rather than \texttt{∂} to convey a derivative.
7.8. USING MAPLE...

7.8.8 ... for Laplace Transforms

In Section 1.5.2, we defined the Laplace transform \( \tilde{f}(s) \) of a function \( f(t) \) by the integral

\[
\tilde{f}(s) = \int_{0}^{\infty} e^{-st} f(t) \, dt
\]  

(7.1)

and showed a number of the properties exhibited by this integral transform. Among other properties, the Laplace transform has the capacity to convert a linear, ordinary differential equation into a linear algebraic equation for the Laplace transform of the solution. Thus, as we shall see in greater detail in Chapter 11, one effective strategy for solving a linear ordinary differential equation is to take its Laplace transform, solve the resulting algebraic equation for that transform, and then invert that transform to return from the \( s \)-space of the transform to the \( t \)-space (time-space) of the original problem. The initial conditions are automatically incorporated in this approach. Note, unfortunately, that inverting the Laplace transform of the solution to find the solution itself is rarely easy.

Within MAPLE, an assortment of integral transforms is provided in the supplementary package \texttt{inttrans}. In particular the function \texttt{laplace} in that package will calculate a Laplace transform and the function \texttt{invlaplace} will invert a transform to return to the original space. We illustrate the use of those functions with the statements

```maple
> with( inttrans):
> laplace( sin(omega*t), t, s );
\omega
\frac{s^2 + \omega^2}{s^2 + \omega^2}
> simplify( invlaplace( %, s, t ) );
\sin(\omega t)
> laplace( exp(-a*t), t, s );
\frac{1}{s + a}
> invlaplace(1/( (s+a)*(s+b) ), s, t );
\frac{e^{-at}}{a-b} + \frac{e^{-bt}}{a-b}
```

Invoke \texttt{inttrans} package. (Terminated with a semicolon, this statement will also list the functions available in the package.)

Calculate the Laplace transform of the function \( \sin(\omega t) \), where \( t \) is the primary variable and \( s \) is the variable in terms of which the transform is to be expressed.

Then, invert and simplify the transform to return to the original function.

Calculate the Laplace transform of the function \( e^{-at} \).

Calculate the inverse Laplace transform of the function \( \frac{1}{(s+a)(s+b)} \).

Restart MAPLE. (See item 6 in Section 7.1.)

7.8.9 ... for Ordinary Differential Equations

Among MAPLE’s strongest suits is its ability to solve ordinary differential equations (ODEs). We will postpone introducing the more sophisticated of those capabilities until Chapter 11, limiting ourselves now to a quick illustration of the simplest of them. We first tell MAPLE the differential equation. For the sake of a specific example, we here choose the motion of an object of mass \( m \) attached to a spring having constant \( k \) and to a dashpot (shock absorber) having damping constant \( b \). The appropriate equation of motion is communicated to MAPLE with the statement

```maple
> deq1 := m*Diff(x(t),t$2) + b*Diff(x(t),t) + k*x(t) = 0;
```

\( \begin{align*}
\text{deq1} & := m \frac{\partial^2 x(t)}{\partial t^2} + b \frac{\partial x(t)}{\partial t} + k x(t) = 0
\end{align*} \)
Here \( \text{Diff}(x(t),t^2) \) specifies the second derivative of \( x \) with respect to \( t \) while \( \text{Diff}(x(t),t) \) specifies the first derivative.\(^{36}\) Next, to solve the equation for \( x \) as a function of \( t \), we submit the statement

\[
> \text{soln} := \text{dsolve}( \text{deq1}, x(t) );
\]

\[
soln := x(t) = C1 e \left( -\frac{1}{2} \frac{b + \sqrt{b^2 - 4km}t}{m} \right) + C2 e \left( -\frac{1}{2} \frac{1}{m} \right)
\]

Here, \( \text{dsolve} \) has used the symbols \( C1 \) and \( C2 \) for the initially undetermined integration constants in the solution to a second-order differential equation. In response to the simple statement we have used, however, MAPLE has blindly written a solution that is mathematically correct but has not recognized that the most convenient form of this solution will depend on the algebraic sign of the quantity under the square root in the two exponents. There are three cases:

1. \( b^2 > 4km \) (overdamped), in which case the square roots are real and both terms have real—though in both cases decaying—exponentials.

2. \( b^2 = 4km \) (critically damped), in which case the two terms are the same and MAPLE has returned only one solution when the general solution should be a linear superposition of two linearly independent solutions. The returned solution is mathematically correct, but we must carefully evaluate the limit as \( b^2 \) approaches \( 4km \) to extract a more transparent form.

3. \( b^2 < 4km \) (underdamped), in which case the square roots are imaginary and the complex exponentials that then appear would more appropriately be written in terms of sines and cosines of real arguments, each multiplied by the decaying exponential \( e^{-bt/2m} \).

To achieve a more appropriate form in the case of underdamped motion, we need to tell MAPLE ahead of time that \( b^2 < 4km \), an end accomplished with the statement

\[
> \text{assume}( b^2 < 4\star k\star m );
\]

Then, the statement we used above to solve \( \text{deq1} \), namely

\[
> \text{soln} := \text{factor}( \text{dsolve}( \text{deq1}, x(t) ) );
\]

will return the result

\[
soln := x(t) = e^{\frac{-b^2 t}{2m^2}} \left( C1 \cos \left( \frac{1}{2} \frac{\sqrt{-b^2 + 4km} t}{m} \right) + C2 \sin \left( \frac{1}{2} \frac{\sqrt{-b^2 + 4km} t}{m} \right) \right)
\]

expressed in terms of the sines and cosines of real arguments. (Similar simpler expressions for the critically and overdamped cases are deduced in the exercises.)

We, of course, are not finished until we have imposed suitable initial conditions on the solution. To do so, we determine \( C1 \) and \( C2 \) to reflect the simple assumption that the motion is started by drawing the oscillator aside and releasing it from rest, i.e., that

\[
x(0) = x_0 ; \quad \frac{dx}{dt}(0) = 0 \quad (7.2)
\]

Then, we find the constraints imposed on the integration constants with the statements

\[^{36}\text{We use the inert form \text{Diff} rather than the evaluated form \text{diff} simply as a safeguard in case we had previously assigned a \( t \)-dependent meaning to the symbol \( x(t) \). Most of the time, we need not be that careful, and the function \text{diff} would work equally well.}\]
7.8. USING MAPLE ...

> eq1 := eval( rhs(soln), t=0 ) = x0;

\[ \text{eq1} \equiv C_1 = x_0 \]

> eq2 := eval( diff(rhs(soln),t), t=0 )=0;

\[ \text{eq2} \equiv -\frac{1}{2} \frac{b^\prime \cdot C_1}{m} + \frac{1}{2} \frac{C_2^2 \sqrt{-b^\prime{}^2 + 4k^\prime{}^2 m}}{m} = 0 \]

and solve the resulting equations for the two integration constants with the statement

> solve( \{eq1,eq2\}, \{C_1,C_2\} );

\[ \{ C_1 = x_0, C_2 = \frac{b^\prime x_0}{\sqrt{-b^\prime{}^2 + 4k^\prime{}^2 m}} \} \]

Finally, we insert these values of the integration constants into the original solution with the statement

> soln := eval( soln, % );

\[ x(t) = e^{-\left(\frac{1}{2} \frac{b^\prime t}{m}\right)} \left( x_0 \cos \left( \frac{1}{2} \frac{\sqrt{-b^\prime{}^2 + 4k^\prime{}^2 m}}{m} t \right) \right) + \frac{b^\prime x_0 \sin \left( \frac{1}{2} \frac{\sqrt{-b^\prime{}^2 + 4k^\prime{}^2 m}}{m} t \right)}{\sqrt{-b^\prime{}^2 + 4k^\prime{}^2 m}} \]

We leave it to the reader to use MAPLE to verify that this solution in fact satisfies the original equation and the imposed initial conditions.

The same result could have been obtained more quickly by specifying the initial conditions in the arguments submitted to \texttt{dsolve} in the first place. The more comprehensive statement

> soln1 := dsolve( \{deq1, x(0)=x0, D(x)(0)=0\}, x(t) );

would have resulted in substantially the same output. Here, the single-character operator \texttt{D} conveys the derivative of the solution to \texttt{dsolve}, i.e., \texttt{D(x)(0)} means the derivative of \texttt{x(t)} with respect to \texttt{t} evaluated at \texttt{t = 0}. Note specifically that the equation and initial conditions have here been presented to \texttt{dsolve} as a \texttt{set}.

Whichever of the three cases applies, we can give the solution a somewhat simpler form by introducing a couple of new quantities. In particular, for the underdamped solution \texttt{soln} above, let us introduce the quantity \[ \gamma = \frac{b}{2m} \]

> soln1 := subs( b=2*m*gamma, soln );

then introduce the quantity \[ \omega \] satisfying \[ km - m^2 \gamma^2 = m^2 \omega^2 \] with the statement

> soln2 := subs( k=m*(omega^2+gamma^2), soln1 );

and finally simplify the result with the statements

> soln3 := simplify(soln2, assume=positive );
We discover in this shorthand notation that

\[ \text{soln3} := x(t) = \frac{e^{-\gamma t}x_0(\cos(\omega t) + \gamma \sin(\omega t))}{\omega} \]

In working its magic, \texttt{dsolve} actually invokes a succession of methods in turn until it finds one that works. We can learn more about the approach \textsc{maple} takes to solving a particular ODE by instructing \textsc{maple} to tell us more detail about what it is doing along the way, especially with respect to the command \texttt{dsolve}. The statement

\[ > \text{infolevel}[\text{dsolve}] := 3: \]

requests that additional detail. To obtain that detail, however, we must reexecute the command that produced the solution with the statement

\[ > \text{dsolve}( \text{deq1}, x(t) ); \]

This time, \textsc{maple} presents the report

\textbf{Methods for second order ODEs:}
--- Trying classification methods
trying a quadrature
checking if the LODE has constant coefficients
<- constant coefficients successful
linear constant coefficient successful

before presenting the solution we obtained earlier. We learn that \textsc{maple} has determined that the equation is a linear equation with constant coefficients and then applied the method that it knows to work in those circumstances.

Much of our manipulation in the above illustration would have been simplified if we had expressed the original equation in dimensionless form \textit{before} seeking its solution. To that end, we might have submitted the statements

\[ > \text{restart}; \]
\[ > \text{deq1} := m*\text{Diff}(x(t),t$2) + b*\text{Diff}(x(t),t) + k*x(t) = 0; \]
\[ > \text{deq1} := m\frac{\partial^2 x(t)}{\partial t^2} + b\frac{\partial x(t)}{\partial t} + kx(t) = 0 \]
\[ > \text{deq2} := \text{subs}( k = m*\text{omega}[0]^2, \text{deq1} ); \]
\[ > \text{deq2} := m\frac{\partial^2 x(t)}{\partial t^2} + b\frac{\partial x(t)}{\partial t} + m\omega_0^2 x(t) = 0 \]

Next, we change the independent variable to \( \tau = \omega_0 t \) by invoking the command \texttt{dchange} from the package \texttt{PDEtools} with the statement

\[ > \text{deq3} := \text{PDEtools}[\text{dchange}](t = \text{tau}/\text{omega}[0], \text{deq2}, \text{[tau]}); \]
\[ > \text{deq3} := m\omega_0^2 \left( \frac{\partial^2 x(\tau,\omega_0)}{\partial \tau^2} \right) + b\omega_0 \left( \frac{\partial}{\partial \tau} x(\tau,\omega_0) \right) + m\omega_0^2 x(\tau,\omega_0) = 0 \]

\footnote{Note that the tildes flagging variables on which assumptions have been imposed have disappeared because the variables subject to explicit assumptions are no longer present in the expression.}

\footnote{Different versions of \textsc{maple} may present different reports.}
In this statement, the first argument specifies the old variable as a function of the new variable, the second argument identifies the differential equation to be transformed, and the third argument—a list (which explains the square brackets)—identifies the new variables. We continue in this simplification by introducing the single symbol $\beta$ for the combination $b/(m\omega_0)$ with the statement

\[
> \text{deq4 := subs( } b = \beta \cdot m \cdot \omega_0[0], \text{deq3 );}
\]

\[
deq4 := m\omega_0^2 \left( \frac{\partial^2}{\partial \tau^2} x(\tau, \omega_0) \right) + \beta m\omega_0^2 \left( \frac{\partial}{\partial \tau} x(\tau, \omega_0) \right) + m\omega_0^2 x(\tau, \omega_0) = 0
\]

Finally, we divide out and cancel the common factor $m\omega_0^2$ and replace $x(\tau, \omega_0)$ with a simpler function $y(\tau)$ by executing the statements

\[
> \text{deq5 := factor( deq4 )/(m*omega[0]^2);}
\]

\[
deq5 := \left( \frac{\partial^2}{\partial \tau^2} x(\tau, \omega_0) \right) + \beta \left( \frac{\partial}{\partial \tau} x(\tau, \omega_0) \right) + x(\tau, \omega_0) = 0
\]

\[
> \text{deq6 := subs( } x(\text{tau,omega}[0])=y(\text{tau}, \text{deq5 };}
\]

\[
deq6 := \frac{\partial^2 y(\tau)}{\partial \tau^2} + \beta \frac{\partial y(\tau)}{\partial \tau} + y(\tau) = 0
\]

At this point, we have swallowed all of the dimensional constants either into a rescaling of the time ($\tau = \omega_0 t$) or into a single dimensionless parameter ($\beta = b/m\omega_0$). We could, of course, rescale what was originally $x(t)$ and is now $y(\tau)$ to some chosen dimensionless quantity, but the equation is linear in this quantity and we can simply interpret the equation as it stands in those terms. We would then solve the equation as before with the statement\(^{39}\)

\[
> \text{assume( } \beta < 2, \beta > 0 \text{ );}
\]

\[
> \text{solln := expand( dsolve( } \{\text{deq6}, y(0)=y[0], D(y)(0)=0\}, y(\text{tau} ) \text{ );}
\]

\[
solln := y(\tau) = \frac{y_0 \beta^{-1} e^{-\frac{1}{2} \beta^{-1} \tau} \sin \left( \frac{1}{2} \sqrt{4 - \beta^{-2}} \tau \right)}{\sqrt{4 - \beta^{-2}}} + y_0 e^{-\frac{1}{2} \beta^{-1} \tau} \cos \left( \frac{1}{2} \sqrt{4 - \beta^{-2}} \tau \right)
\]

Restart MAPLE. (See item 6 in Section 7.1.

We leave it to the reader to show—perhaps using MAPLE—that this solution is a dimensionless form of the dimensional result obtained earlier in this section.

We have, of course, mentioned only the command \texttt{dsolve} as a function for solving ordinary differential equations, and we have introduced the command \texttt{dchange} in the package \texttt{PDEtools} for changing variables in differential equations and the command \texttt{infolevel} for controlling the amount of information MAPLE provides about its workings behind the scenes. We have also seen additional contexts in which the commands \texttt{assume}, \texttt{eval}, \texttt{rhs}, \texttt{solve}, \texttt{subs}, \texttt{simplify}, \texttt{factor}, and \texttt{expand} are useful. Your attention is drawn to the packages \texttt{DEtools} (and especially to the function \texttt{odeadvisor} in that package) and to the (rest of the) package \texttt{PDEtools}, which contain additional functions for obtaining solutions to differential equations and for displaying their characteristics. You should make it a point to read about these functions in the MAPLE manuals.

\(^{39}\)Here, the assumption $\beta < 2$ is equivalent to the assumption $b/m\omega = b/(m \sqrt{k/m}) < 2$ or $b^2/mk < 4$ or $b^2 < 4mk$. Thus, the \textit{dimensionless} inequality $\beta < 2$ here is exactly equivalent to the \textit{dimensional} inequality $b^2 < 4km$ that appeared earlier in this section.
7.8.10 ... for Vector Calculus

By including the package `VectorCalculus` (see Section 7.9), MAPLE makes available several commands to facilitate vector calculus. Once this package has been added with the statement

\[
> \text{with( VectorCalculus ):}
\]

numerous commands, including in particular the commands

- `Gradient`, `Divergence`, `Curl`, and `Laplacian` for the gradient, divergence, curl, and Laplacian, respectively. As illustrated in the rest of this section, the `Gradient` and `Laplacian` adopt one way to convey the coordinate system and associated variable names while the `Divergence` and `Curl` utilize a different approach.
- `SetCoordinates` and `GetCoordinates` for specifying and revealing the coordinate system.
- `VectorField` for specifying the components of a vector field.

We begin this illustration by defining scalar quantities in each of the three common coordinate systems with the statements

\[
> \text{scar := } x^2+y^2+z^2;
\]

\[
\text{scar := } x^2+y^2+z^2
\]

... cylindrical coordinates.

\[
> \text{scyl := } r^2+z^2;
\]

\[
\text{scyl := } r^2+z^2
\]

... spherical coordinates.

\[
> \text{ssph := } r^2;
\]

\[
\text{ssph := } r^2
\]

Define a scalar in Cartesian coordinates.

Then, we evaluate the gradients with the statements

\[
> \text{Gradient(} \text{scar}, \text{'}cartesian'\[x,y,z]\) ;
\]

\[
\text{Gradient(} \text{scar}, \text{'}cartesian'\[x,y,z]\) ;
\]

Evaluate \(\nabla\) in Cartesian coordinates.

\[
> \text{Gradient( scyl, } \text{'}cylindrical'\[r,phi,z]\) ;
\]

\[
\text{Gradient( scyl, } \text{'}cylindrical'\[r,phi,z]\) ;
\]

... cylindrical coordinates.

\[
> \text{Gradient( ssph, } \text{'}spherical'\[r,theta,phi]\) ;
\]

\[
\text{Gradient( ssph, } \text{'}spherical'\[r,theta,phi]\) ;
\]

... spherical coordinates.

Here, \(\hat{e}_r\) represents a unit vector in the coordinate direction conveyed by the subscript. Continuing, we evaluate the Laplacian with the statements

\[
> \text{Laplacian(} \text{scar}, \text{'}cartesian'\[x,y,z]\) ;
\]

\[
\text{Laplacian(} \text{scar}, \text{'}cartesian'\[x,y,z]\) ;
\]

Evaluate \(\nabla^2\) in Cartesian coordinates.

\[
> \text{Laplacian( scyl, } \text{'}cylindrical'\[r,phi,z]\) ;
\]

\[
\text{Laplacian( scyl, } \text{'}cylindrical'\[r,phi,z]\) ;
\]

... cylindrical coordinates.

\[
> \text{Laplacian( ssph, } \text{'}spherical'\[r,theta,phi]\) ;
\]

\[
\text{Laplacian( ssph, } \text{'}spherical'\[r,theta,phi]\) ;
\]

... spherical coordinates.

In each case, the second argument specifies both the coordinate system and the variable to be used for the coordinates. Among the multitude of available two-dimensional coordinate systems are cartesian, polar, elliptical, and hyperbolic; among the multitude of available three-dimensional systems are cartesian, cylindrical, spherical, conical, and bispherical. Note also that the coordinate system assumed by default is cartesian. The statement

\[
\text{SetCoordinates();}
\]

will report the current default system while the statement

\[
\text{SetCoordinates( } \text{'}\langle\text{Coordinate Name}\rangle\text{'\[\langle\text{Variables}\rangle\] );}
\]

will set the coordinates to the specified system.
will set the default to the specified name and variables.

Functions evaluating the divergence and the curl are a bit more tricky. First the single input to each of these commands must be a vector field, which we define in the three common coordinate systems with the statements

\[
\begin{align*}
&\texttt{vcar} := \text{VectorField}( \langle x, y, z \rangle, \text{’cartesian’}[x, y, z] ); \\
&\quad {vcar} := x \hat{e}_x + y \hat{e}_y + z \hat{e}_z \\
&\texttt{vcyl} := \text{VectorField}( \langle r, 0, z \rangle, \text{’cylindrical’}[r, \phi, z] ); \\
&\quad {vcyl} := r \hat{e}_r + z \hat{e}_z \\
&\texttt{vsph} := \text{VectorField}( \langle r, 0, 0 \rangle, \text{’spherical’}[r, \theta, \phi] ); \\
&\quad {vsph} := r \hat{e}_r
\end{align*}
\]

In this case, the coordinate system and variables applicable to each field must be explicitly stated as attributes of the field. The divergence of each of these fields is now evaluated with the statements

\[
\begin{align*}
&\texttt{Divergence}( \texttt{vcar} ); \\
&\quad \text{Evaluate } \nabla \cdot \text{ in Cartesian coordinates.} \\
&\texttt{Divergence}( \texttt{vcyl} ); \\
&\quad \text{... cylindrical coordinates.} \\
&\texttt{Divergence}( \texttt{vsph} ); \\
&\quad \text{... spherical coordinates.}
\end{align*}
\]

and, finally, the curl of each is evaluated with the statements

\[
\begin{align*}
&\texttt{Curl}( \texttt{vcar} ); \\
&\quad \text{Evaluate } \nabla \times \text{ in Cartesian coordinates.} \\
&\texttt{Curl}( \texttt{vcyl} ); \\
&\quad \text{... cylindrical coordinates.} \\
&\texttt{Curl}( \texttt{vsph} ); \\
&\quad \text{... spherical coordinates.}
\end{align*}
\]

Evidently, when all components of the curl are zero, MAPLE reports only the first component.

MAPLE can also evaluate these vector derivatives when the dependence in the first argument on the coordinates is not explicit. For example,\(^{40}\)

\[
\begin{align*}
&\texttt{Laplacian}( f(r, \phi, z), \text{’cylindrical’}[r, \phi, z] ); \\
&\quad f_{rr}(r, \phi, z) + \frac{1}{r} f_r(r, \phi, z) + \frac{1}{r^2} f_{\phi\phi}(r, \phi, z) + f_{zz}(r, \phi, z)
\end{align*}
\]

Remember that the independence of the mixed derivatives on the order in which the derivatives are evaluated is critical to use of this notation.

We have, of course, mentioned only the commands \texttt{Gradient}, \texttt{Divergence}, \texttt{Laplacian}, and \texttt{Curl} for working with coordinate dependent vectors and scalars, the command \texttt{VectorField} for creating a vector field with the proper attributes, and the commands \texttt{SetCoordinates} and \texttt{GetCoordinates} for defining and revealing the default coordinate system and variable names. The package \texttt{VectorCalculus} includes a wide variety of commands for a variety of other tasks involving manipulations with vectors and coordinate systems. You should make it a point to issue the statement that loads this package, read the resulting list of commands contained in the package, and look at the documentation on any of these commands that attract your attention.

\(^{40}\)For compactness, we have rearranged MAPLE’s output and used subscript notation for partial derivatives rather than the more cumbersome notation reported by MAPLE.
7.9 Packages

The packages LinearAlgebra introduced in Section 7.8.6, inttrans introduced in Section 7.8.8, and DEtools and PDEtools introduced in Section 7.8.9 are but four of numerous packages that can be read into MAPLE's workspace to expand or modify the features available in the default version when MAPLE is first launched. Use of the command PckgCommand from the package PckgName requires either (1) explicit specification of the full “path” with the statement

> PckgName[PckgCommand]( ... );

when the command is invoked, (2) prior incorporation of the specific command with the statement

> with( PckgName, PckgCommand );

or (3) prior incorporation of all the commands in the package with the statement

> with( PckgName );

With the second and third of these methods, the command itself can subsequently be invoked with the simple statement

> PckgCommand( ... );

A list of available packages will be displayed in response to the statement ?index,packages,41 information about a particular package will be displayed in response to the statement ?PckgName, and information about a particular command in a particular package will be displayed in response to the statement ?PckgName,PckgCommand. Your attention is drawn especially to the packages DEtools, group, inttrans, LinearAlgebra, numapprox, PDEtools, plots, plottools, Student, sumtools, tensor, Units, and VectorCalculus.

7.10 Loops, Logical Expressions, and Conditionals

Among the most ubiquitous programming structures is the loop, which provides a means by which a statement or block of statements can be executed some number of times, typically with small changes controlled by a loop index. Several such structures are available in MAPLE. The simplest is the for...do...end do loop, which we have already met in Sections 7.8.2, 7.8.4, and 7.8.5. Briefly, the statements

> x := [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ];
  x := [0,0,0,0,0,0,0,0,0,0]
> for i from 1 by 2 to 10 do x[i] := i^2: end do:
> x;
  [1, 0, 9, 0, 25, 0, 49, 0, 81, 0]

create a 10-element list of zeros and then assign the squares of the odd integers to the odd elements of that list. In this structure, the component by can be omitted, in which case the index—here i—will be incremented by 1 at each pass through the loop.

A more general, flexible loop structure is also available, though in some instances termination of the loop requires that the block of instructions in the loop must assume responsibility for changing the condition to a value that will stop the loop. The general syntax of these loops is

41Note that there is no space after the comma in this statement.
More specifically, these loops are illustrated in the statements

```plaintext
> x := [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]:
> for i from 1 by 2 while i < 11 do x[i] := i^2: end do:
> x;
[1, 0, 9, 0, 25, 0, 49, 0, 81, 0]
> i := 1:
> while i < 11 do x[i] := i^3: i := i+2: end do:
> x;
[1, 0, 27, 0, 125, 0, 343, 0, 729, 0]
```

As here suggested,

- The for, from, by, to, and while components can be omitted or included in any order (except that the for component, if present, must be first).\(^{42}\)
- If the to component is omitted, then the block of statements must include statements that ultimately toggle (condition) such that the loop terminates.

Note that, in all cases, the block of statements in this structure is enclosed between do and end do and the statements are separated by a colon or a semicolon, depending on whether output is to be produced or suppressed.

Finally, a word about composite conditions: The illustrations above made use of (single) conditions like \(i > 9\), which will evaluate to true or false depending on the value of \(i\). Indeed, the MAPLE function evalb, for example,

```plaintext
> i := 5:
> [ evalb( i > 10 ), evalb( i < 10 ) ];
[false, true]
```

allows the testing of a logical condition. The six logical operators =, <, >, <=, >=, and <> can be used for test for equal, less than, greater than, less than or equal, greater than, and unequal.

In some situations, one may need to (1) combine two or more conditions into a composite condition or (2) negate a condition. MAPLE makes available the logical operators and, or, xor (exclusive or), and not to facilitate combining conditions and negating conditions. More specifically, with \(\langle C1 \rangle\) and \(\langle C2 \rangle\) individual conditions

- \((\langle C1 \rangle \text{ and } \langle C2 \rangle)\) will be true if \(\langle C1 \rangle\) and \(\langle C2 \rangle\) are both true and false otherwise,
- \((\langle C1 \rangle \text{ or } \langle C2 \rangle)\) will be true if either \(\langle C1 \rangle\) or \(\langle C2 \rangle\) is true or both \(\langle C1 \rangle\) and \(\langle C2 \rangle\) are true and false otherwise, and
- \((\langle C1 \rangle \text{ xor } \langle C2 \rangle)\) will be true if either \(\langle C1 \rangle\) or \(\langle C2 \rangle\) is true and false if neither condition is or both are true, and
- \(\text{not } \langle C1 \rangle\) will be true if \(\langle C1 \rangle\) is false and false if \(\langle C1 \rangle\) is true.

For example, the coding

\(^{42}\)Omission of the from or by components will trigger the default values from 1 or by 1.
illustrates a composite condition controlling the while statement. Note that parentheses around the composite condition can in this case be omitted, but they—and perhaps parentheses around some internal pieces—may be necessary in some cases.

Logical conditions appear not only to control loops but also to structure branches in a sequence of statements. As with most programming languages, MAPLE also possesses if/then/else constructs, though the else clause can be omitted if there is in that case nothing to be done. Thus, for example, the statement

```maple
for i from 1 to 10 do if x[i] < 0 then x[i] := -x[i]: end if: end do:
```

will replace each negative element in a list \( x \) with the corresponding positive value, the statement

```maple
for i from 1 to 10 do if x[i] > 10 then x[i] := 10: end if: end do:
```

will replace all values greater than 10.0 with the value 10.0, and the statement

```maple
if a >= 0 then b := a else b := -a; end if;
```

will set \( b \) equal to the absolute value of \( a \) (though the function \texttt{abs} will do so more easily). Multiple statements in either the then clause or the else clause will be separated by commas and enclosed within parentheses, as in

```maple
if \langle \text{condition}\rangle \text{then} \langle \text{statement}\rangle, \langle \text{statement}\rangle, \ldots \rangle
else \langle \text{statement}\rangle, \langle \text{statement}\rangle, \ldots \rangle; \text{end if}
```

For selection among more than two options, the statement would have the form

```maple
if \langle \text{condition}\rangle \text{then} \langle \text{block of statements}\rangle
elif \langle \text{condition}\rangle \text{then} \langle \text{block of statements}\rangle
::
else \langle \text{block of statements}\rangle;
\text{end if}
```

All conditions may be single or composite.

As is the case in most languages, loops and conditional structures can be nested, though constructing the syntax correctly can sometimes be challenging.

### 7.11 Command Files

In addition to responding to statements supplied interactively at its prompt, MAPLE can read statements from a command file. To begin, we create a text file containing the desired statements just as they would be typed interactively. If the file is stored in the default directory,\(^{43}\) execution of the command(s) contained within it would then be specified by typing the statement

---

\(^{43}\)See Section 7.17.2.
read "FileName";

in response to MAPLE’s prompt for input. There is no default file type.\footnote{The author is accustomed to using the filetype .mpl. Only the type .m must be avoided, since that filetype will convey to the command read that the file contains information in MAPLE’s internal (binary) format—i.e., that the file is a file created by MAPLE with the command save.} Except when the file is in the default directory, the full path must be given to supplement the file name and type. Output generated by commands in the file will be displayed on the screen, but—by default—the commands themselves will not be displayed. The MAPLE environment available after the command file has been executed by this means is identical to the environment that would have existed had the statements been entered interactively.

Sometimes, creation of a command file facilitates repeated use of quantities not defined in standard MAPLE. Use of such a file can also facilitate the debugging of a sequence of statements, since—when an error is reported—we need only edit the file appropriately and resubmit it to MAPLE; repeated typing of the correct parts of the sequence is not then necessary. We could, for example, create the file testmultiply.mpl containing the statements

```maple
# Command file testmultiply.mpl

# Enter matrix
Mat1 := Matrix( 1..2,1..2, [[1,2], [3,4]]);
# Evaluate transpose
Mat2 := LinearAlgebra[Transpose](Mat1);
# Evaluate matrix product
LinearAlgebra[Multiply]( Mat1, Mat2 );
# Multiply element by element
Arr1 := Array( [[1,2],[3,4]] );
Arr2 := Array( [[1,3],[2,4]] );
ArrProd := Arr1*Arr2;
```

(The symbol # introduces a comment that continues from the position of the symbol to the end of the line.) Then, assuming that this file has been stored in the default directory,\footnote{See again Section 7.17.2.} we would execute the statements it contains by submitting to MAPLE the statement

```maple
> read "testmultiply.mpl";
```

All output produced by the statements will be displayed on the screen but, with MAPLE’s default configuration, none of the statements or comments in the file will be displayed.

At times, we might wish to see the statements and comments as the command file executes. To achieve that behavior, we simply set the echo characteristic of the interface between the program and the display device away from its default value of 1 to the value 2 with the statement

```maple
> interface( echo=2 );
```

Once this statement has been executed, the output from the above read statement will include a display of the statements and comments in the file. (The statement interface( echo=1 ) will restore the default behavior.)

Suppose, as a second example, that we frequently had two three-component lists defining two three-dimensional vectors and that we wanted to be able easily to evaluate their cross and dot products. We might create a command file containing the lines
CHAPTER 7. INTRODUCTION TO MAPLE

# crossdot.mpl

# Command file to define functions to evaluate the cross
# and dot products of two three-component lists.

cross := (a::list,b::list) ->
  a[3]*b[1] - a[1]*b[3],

dot := (a::list,b::list) -> a[1]*b[1] + a[2]*b[2] + a[3]*b[3];

Here, beyond the comments, the first four lines—actually a single statement—define the function
\texttt{cross}, which takes two three-component lists as arguments and returns a three component list whose
components are those of the cross product of the two vectors. The last line defines the function \texttt{dot},
which takes the same arguments but returns a scalar value equal to the dot product of those two
lists treated as three-dimensional vectors. In both cases, we have exploited the MAPLE operator
\texttt{::} that allows us to declare the data type of particular variables, thereby resulting in smooth error
messages rather than mystery or worse in the event an entry of the wrong data type is supplied.

This file is named \texttt{crossdot.mpl} and can be accessed in the directory $\texttt{HEAD/maple}$.

Thus, the
statement

\texttt{> read "$\texttt{HEAD/maple/crossdot.mpl}";}

will load this file (silently when the the variable \texttt{echo} has its default value) defining the two func-
tions \texttt{dot} and \texttt{cross} for subsequent use. For example, once these functions have been defined, the
statements

\texttt{> cross( [1,2,3], [4,5,6] );}
\texttt{[-3,6,-3]}
\texttt{> dot( [1,2,3], [4,5,6] );}
\texttt{32}

will evaluate the cross and dot products of the indicated three-component lists.

Be aware, as well, that the package \texttt{LinearAlgebra} contains more robust and more carefully
structured functions \texttt{CrossProduct} and \texttt{DotProduct} to achieve the ends provided by the functions
defined in \texttt{crossdot.mpl}. A means to obtain listings of the MAPLE codings in those two functions
involves the command \texttt{print} and is described in Section 7.18.

7.12 High-Resolution Graphing

We illustrate the production of high resolution graphics by plotting the solution for the damped
harmonic oscillator obtained in Section 7.8.9. First, we reclaim the solution with the statement

\texttt{>, soln3 := x(t) = x0 \times \exp(-gamma*t) \times (gamma*sin(omega*t)/omega + cos(omega*t));}

\texttt{soln3 := x(t) = e^{-gamma*t} \times (gamma*sin(omega*t)/omega + cos(omega*t))/omega}

\footnote{Remember that the symbol $\texttt{HEAD}$ must be translated into the appropriate path as described in the \textit{Local Guide}. All files referred to in this chapter are available in the directory $\texttt{HEAD/maple}$.}
Further, to simplify subsequent expressions and anticipate interest in velocity as well as position, we extract the position as an expression and evaluate also the velocity with the statements

\[ \text{soln3} := \text{rhs}\left(\text{soln3}\right); \]
\[ \text{soln3dot} := \text{diff}\left(\text{soln3}, \text{t}\right); \]

Then, recognizing that \( \gamma \) and \( \omega \) are not independent \( (\omega^2 = k/m - \gamma^2) \) and hence that assigning values to \( \gamma \) and \( \omega \) in effect fixes \( k/m \), we use the statements\(^{47,48}\)

\[ \text{unprotect}\left(\text{gamma}\right); \]
\[ \gamma, \omega, x_0 := 0.1, 1.0, 1.0; \]

to assign specific values to \( \gamma \), \( \omega \), and \( x_0 \). Finally, we create a graph showing position and velocity with the statement

\[ \text{plot}\left(\left[\text{soln3}, \text{soln3dot}\right], \text{t}=0.0..20.0, \text{y}=-1.0..1.0; \right); \]

which plots the identified expressions over the intervals \( 0.0 \leq t \leq 20.0 \) in \( t \) and \( -1.0 \leq y \leq 1.0 \) in \( y \) (the vertical coordinate). The graph is produced as a coarse character plot if we are working in the MAPLE CLI and as a smooth plot with—by default—at least 200 points\(^{49}\) distributed over the range of the independent variable in the MAPLE GUI.

Alternatively, we could modify the resulting graph by exploiting a few of the many keywords supported by the command \texttt{plot}. The statement

\[ \text{plot}\left(\left[\text{soln3}, \text{soln3dot}\right], \text{t}=0.0..20.0, \text{y}=-1.0..1.0, \text{numpoints}=100, \right. \]
\[ \left. \text{color}=\text{black}, \text{labels}=\left["t","x,v"\right], \text{title}=\text{"Damped Oscillator"}, \right. \]
\[ \left. \text{thickness}=3, \text{linestyle}=\left[1,3\right], \text{labelfont}=\text{\{TIMES,ROMAN,16\}}, \right. \]
\[ \left. \text{titlefont}=\text{\{TIMES,ROMAN,20\}}, \text{axesfont}=\text{\{TIMES,ROMAN,14\}}, \right. \]
\[ \left. \text{labeldirections}=\text{\{HORIZONTAL,VERTICAL\}} \right); \]

for example, will produce the same graph but with

- a minimum of 100 points spaced over the range of the independent variable (See footnote 49.),
- all graphs displayed in black (We anticipate output to a monochrome printer.),
- the indicated labels on the axes,
- the indicated title on the graph as a whole,
- all expressions (but not the axes) displayed with a line thickness heavier than the default,
- the first expression drawn with linestyle 1 (solid) and the second with linestyle 3 (dashed),
- the axis labels (\texttt{labelfont}), the title (\texttt{titlefont}), and the tic labels (\texttt{axesfont}) drawn in times, Roman type of the indicated sizes (in points), and
- the directions of the labels on the axes set to horizontal (the default) on the horizontal axis and vertical (not the default) on the vertical axis.

These (and numerous other) keywords to the command \texttt{plot} are fully described in the on-line help message accessed with the statement \texttt{?plot,options}. The graph resulting from this last statement is shown in Fig. 7.1. Note that the values labeled by default on each axis depend on the version of MAPLE.

\(^{47}\)The variable name \texttt{gamma} is protected, being by default used for the Euler constant. To use it for another purpose, we must begin by unprotecting it.

\(^{48}\)We must, of course, remember to restore the standard meaning of the symbol \texttt{gamma} when we are finished. Note, however, that our override of that standard meaning will not survive exiting from the MAPLE session altogether. Thus, we need not fear that failure to restore that meaning at the end of this session will “contaminate” the next session.

\(^{49}\)The default in some earlier versions of MAPLE was 50 points distributed over the range of the independent variable. MAPLE16 uses an adaptive approach with—by default—at least 200 points in that interval.
Figure 7.1: Graph of position and velocity for a damped oscillator with $\gamma = 0.1$, $\omega = 1.0$, $x(0) = x_0 = 1.0$, and $v(0) = 0.0$. Since the solid and dashed lines produced on the screen are not distinct in this print out, we must look to the initial values to identify which curve shows $x(t)$, which shows $v(t)$.

We could explore other features of this motion by, for example, changing the value of $\gamma$ and reploting the graph with the statements

```maple
> gamma := 0.2:
> plot( [soln3, soln3dot], t=0.0..20.0, y=-1.0..1.0, numpoints=100,
>     color=black, labels=["t","x,v"], title="Damped Oscillator",
>     thickness=3, linestyle=[1,3], labelfont=[TIMES,ROMAN,16],
>     titlefont=[TIMES,ROMAN,20], axesfont=[TIMES,ROMAN,14],
>     labeldirections=[HORIZONTAL,VERTICAL] );
```

or, for these new values of the parameters, we could produce a graph of the trajectory in the phase plane ($soln3dot$ versus $soln3$) with the statements

```maple
> plot( [soln3, soln3dot, t=0.0..20.0], x=-1.0..1.0, y=-1.0..1.0,
>     color=black, thickness=3, labels=["x","v"], scaling=CONSTRAINED );
```

Here, the new keyword `scaling`, whose allowed values are `CONSTRAINED` and `UNCONSTRAINED` (the default), permits us to tell MAPLE to use equal scales on both axes so that, for example, circles will plot as circles. The resulting graph is shown in Fig. 7.2, though again the values labeled by default on each axis depend on the version of MAPLE.

MAPLE can, of course, also produce surface plots and contour plots in several different ways, though the functions for doing so are part of the package `plots`. The simplest and quickest such plots are produced with statements like

```maple
> mode := sin( 2*Pi*x ) * sin( 3*Pi*y );
mode := sin(2\pi x)sin(3\pi y)
```
7.12. HIGH-RESOLUTION GRAPHING

Figure 7.2: Graph of velocity versus position for a damped oscillator with $\gamma = 0.2$, $\omega = 1.0$, $x(0) = 1.0$, and $v(0) = 0.0$. 

```maple
> lvls := [-0.9,-0.8,-0.7,-0.6,-0.5,-0.4,-0.3,-0.2,-0.1,0.0, 
> 0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9]:
> plot3d( mode, x=0.0..1.0, y=0.0..1.0, thickness=3 );
> plots[contourplot]( mode, x=0.0..1.0, y=0.0..1.0, 
> color=black, thickness=3, contours=lvls, scaling=CONSTRAINED );

Restart MAPLE. (See item 6 in Section 7.1.)

The first line here defines the expression to be displayed. Subsequent lines set values used in specifying options to the commands and, ultimately, produce both a surface plot and a contour map. By default, the expression `mode` is evaluated on a $25 \times 25$ grid of points equally spaced over the ranges of $x$ and $y$, though the keyword `grid` provides a means to override that default. The resulting graphs are shown in Figs. 7.3 and 7.4.

Another feature of the `plots` package embodies an ability to display text at specified points in the graphical display. For example, to produce Fig. 7.1 with, however, the added the labels ‘$x(t)$’ at the point [2.5, 0.8] and ‘$v(t)$’ at the point [6.5, 0.7], we might use the statements

```maple
> with(plots):
> soln3 := x0 * exp(-beta*t) * (beta*sin(omega*t)/omega + cos(omega*t));
> soln3dot := diff( soln3, t );
> beta, omega, x0 := 0.1, 1.0, 1.0;
> gr := plot( [soln3, soln3dot], t=0.0..20.0, y=-1.0..1.0, numpoints=100, 
> color=black, labels=["t","x,v"], title="Damped Oscillator", 
> thickness=3, linestyle=[1,3], labelfont=[TIMES,ROMAN,16], 
> titlefont=[TIMES,ROMAN,20], axesfont=[TIMES,ROMAN,14], 
> labeldirections=[HORIZONTAL,VERTICAL] );
> lab1 := textplot( [2.5,0.8,"x(t)", font=[TIMES,ROMAN,14]]):
> lab2 := textplot([6.5,0.7,"v(t)", font=[TIMES,ROMAN,14]]):
> display(gr, lab1, lab2);
```
Figure 7.3: Surface plot of the function $z(x, y) = \sin(2\pi x) \sin(3\pi y)$.

Restart MAPLE. (See item 6 in Section 7.1.)

Here, we suppress output in the creation of the graph labeled gr and the two labels lab1 and lab2 before invoking the command display that actually produces the graph.

As may already have been evident, MAPLE is able to produce graphical output either “inline” (i.e., right in the worksheet in the GUI) or in a separate window created when the graph itself is created. The default behavior depends on the platform and on the version of MAPLE. Either method, however, is available on both platforms; we simply use one or the other of the statements

```
plotsetup( inline );
plotsetup( window );
```

to select the desired format. Whichever plot setup is in use, the MAPLE prompt returns after the graph has been displayed.

We have, of course, only mentioned a few [plot, plot3d, and contourplot (in the package plots)] of the functions and only some (numpoints, color, labels, title, thickness, linestyle, labelfont, titlefont, axesfont, labeldirections, scaling, and contours) of the keywords that modify the behavior of these functions. Numerous other functions are contained within the package plots. You should make it a point to read about these functions and the associated keywords in the MAPLE manuals.

### 7.13 Making Hard Copy . . .

The method used to make hard copy of MAPLE worksheets and output depends on whether the CLI or the GUI is in use; whether the material to be output is textual or graphical; whether the graphical material is inline or in a separate window; whether the output is to go directly to a printer or to a PostScript, ASCII, HTML, or ... file; and whether we are working on a UNIX, Windows, or Macintosh platform. In the following subsections we address each of several possibilities in turn. In
7.13. MAKING HARD COPY . . .

all cases when a file name is specified, the file will be located in the current default directory (see Section 7.17.2) unless a full path is included in the file name.

7.13.1 . . . of Text and Character Graphics in the CLI

In the CLI, text and inline graphics are constructed entirely of characters. In many operating systems, printing of information displayed in the CLI is most simply accomplished by using the cutting and pasting capabilities of the operating system to transfer that character information into a text editor. The text can then be edited if desired and the file can be saved and sent to an available printer with the standard commands to the operating system (or perhaps by selecting appropriate options from a menu available in the editor).

Alternatively, the MAPLE statement writeto( "FileName" ), once executed, will suppress display of all following output on the screen, directing that output instead to the specified file. The command prompt and the input statements will appear on the screen but will also be directed to the file unless echoing has been set to zero with the statement interface( echo=0 ). The statement appendto( "FileName" ) behaves in a similar way but appends to an existing file instead of overwriting an existing file. Closing the file and restoring the normal behavior of the terminal is achieved by executing the statement writeto( terminal ). Note the absence of quotation marks in this last statement!50

7.13.2 . . . of High-Resolution Graphs from the CLI

Even though the (inline) graphics produced by plotting commands in the CLI will be character graphs, commands issued in the CLI can write high-resolution graphics into files for subsequent printing. We simply need to inform MAPLE of the form of the driver it is to use in creating the file, the name it is to give the file, and any controlling options. For example, the statement

50Note that writeto and appendto will also work in the GUI, but high-resolution graphics will not be adequately transcribed.
plotsetup( ps, plotoutput="trial.ps", 
    plotoptions='portrait,noborder' );

will tell MAPLE to use the PostScript driver, write the output to the file trial.ps in the default directory, use portrait orientation, and suppress the border. (Note the direction of the single ticks enclosing the value of plotoptions: both are forward tics.) The output of subsequent plot commands will then be constructed in the specified format and directed to the identified file; no output from those commands will appear on the screen. To return to the default (character) presentation of graphical displays, we would use the statement

plotsetup( char ) or plotsetup( default )

Alternatively, we can create high-resolution on-screen graphics in the CLI by selecting the maplet driver with the command

plotsetup( maplet );

In response to a subsequent plot command, MAPLE will then open a new plot window on the screen and display a high-resolution graph in that window. The menu in the top bar of that plot window offers several ways to edit the display. In particular, the FILE menu in the top bar gives access to routines for writing (exporting) the graph to a file in any of several formats. This file can, of course, then be printed or incorporated in larger documents. Note that an existing plot window must be closed before the CLI will respond to commands beyond the plot command.

7.13.3 . . . of Text and Inline Graphics in the GUI

For generating hard copy from the GUI, two items in the FILE menu are available. Selecting ‘Print...’ from the FILE menu will bring up the Print window in which we can select (or specify) the desired printer. Either in tabs or via a PROPERTIES menu, this window allows specification of a variety of options (paper size, page range to be printed, orientation—portrait or landscape—on the page, scaling, ...). The selected pages or the entire contents of the current worksheet (text and graphics) will be directed to the specified printer.

To achieve greater flexibility in output format, select ‘Export As...’ from the FILE menu to bring up a pop-up window in which we can in a dropdown menu select one of several file types (HTML, PDF, \LaTeX, Plain Text (i.e., ASCII), RTF, ...) and, either in that same window or in a browser that appears when the file type has been selected, specify the file name and directory for the file to be created. If ‘Plain Text’ is selected, the graphics will not be included.

7.13.4 . . . of Window Graphics in the GUI

Graphical output that has been directed to a window separate from the window in which the current worksheet exists can be directed to a file in a more convenient way. In MAPLE16, we simply

---

51 Depending on the operating system, the printer may be selected from a dropdown menu or specified by typing the specific printer command in a text entry box.

52 This window also allows specification that the output be directed to a file, in which case we supply the destination directory and file name in a text entry box, which may not appear until the button requesting the print to be created has been clicked. This route to a file, however, offers no control over the file type produced (.prn in windows, .ps in Unix, .pdf in the Macintosh OS).

53 By setting the plot driver with the statement plotsetup(window).

54 In other versions of MAPLE, the procedure may be slightly different but will begin by clicking MR in the plot window.
7.14. OUTPUT IN LATEX FORMAT

- Move the cursor into the graph window.
- Click MR.
- Move the cursor to ‘Export→’.
- Move the cursor to and click ML on the appropriate file type (most likely ‘Encapsulated PostScript’) in the menu that pops up.
- Specify the desired file name and directory in the resulting browser.
- Click on ‘Save’.

The file will be created.

7.14 Output in \LaTeX Format

A particularly useful feature of MAPLE is its ability to write \LaTeX versions of expressions to the screen or into a file. Full details are laid out in the MAPLE manuals. In brief, the statement

\texttt{latex(expr1);} 

for example, converts the specified expression into \LaTeX format and displays it on the screen (from where standard cut and paste operations—when they are available—can move the description to wherever we might want it). Alternatively, the statement

\texttt{latex(expr1, "FileName");} 

will create a new (ASCII) file and write the \LaTeX description of the indicated expression into that file, \textit{overwriting} any existing file by the same name. To \textit{append} the \LaTeX description of the expression to an existing file, the more elaborate statement

\texttt{latex(expr1, "FileName", 'append');} 

is necessary. Once created, this file can, of course, be edited or incorporated into other \LaTeX documents.

Automatic translation of arbitrary expressions into \LaTeX is a daunting task. While fairly sophisticated, the MAPLE translator is not perfect. Edits will often be necessary to “correct” the translator’s glitches. In particular, the translator writes only the \LaTeX description of the equation to the screen or into the file. It does not place the equation in a \LaTeX equation environment or otherwise embellish the description with additional \LaTeX constructs that may be necessary for creation of the ultimate document. These will have to be supplied through manual editing of the file created by MAPLE. Conveniently, variables named \texttt{gamma} and \texttt{GAMMA}, for example, will be properly translated into \texttt{\gamma} and \texttt{\Gamma}.

Even more generally, if we have created a MAPLE notebook in the GUI, we can output that notebook in its entirety (including the textual sections) into a \LaTeX file by selecting ‘Export as’ from the \textit{File} menu and then ‘\LaTeX’ in the resulting pop-up menu. The resulting file is complete and ready to be processed through \LaTeX. Be aware, however, that \LaTeX files written by MAPLE make use of several MAPLE-specific \LaTeX commands which are defined in the MAPLE-supplied \LaTeX package named \texttt{maple2e.sty} or \texttt{maplestd2e.sty} (depending on the version of MAPLE), which will be invoked with the simple statement

\texttt{\usepackage{maple2e} \quad \text{or} \quad \usepackage{maplestd2e}}
in the preamble of the \LaTeX source file. The file defining this package (and several other files—mapleenv.sty, maplestyle.sty, mapleutil.sty, mapleenv.def, mapleplots.sty, and mapletab.sty—to which the primary file maple2e.sty or maplestd2e.sty may refer) must be accessible on your computer system before a MAPLE-produced \LaTeX source file can be processed and printed. Further information about this issue is contained in the MAPLE manuals.

7.15 Animation

The command animate in the plots package provides a facility to create an animation in two or three dimensions on a single parameter. More specifically, the statement

\begin{verbatim}
plots[animate](plot, [sin(Pi*x)*cos(2*Pi*t), x=0.0..1.0], t=0.0..1.0)
\end{verbatim}

will create and store the frames for an animated display of the fundamental mode of a vibrating string fixed at both ends while the statement

\begin{verbatim}
plots[animate](plot3d, [sin(2*Pi*x)*sin(3*Pi*y)*cos(2*Pi*t),...
x=0.0..1.0, y=0.0..1.0], t=0.0..1.0)
\end{verbatim}

will create and store the frames for an animated display of the 23 mode of oscillation of a square membrane. Additional options for plot and plot3d can be included to further structure the display.

The above statements simply create the frames for the animation. Actually playing the animation requires right clicking on the graph and selecting ‘Play’ from the ANIMATION→ menu in the resulting pop-up menu. That menu also offers, among others, the option to play the animation forwards, backwards, or continuously. Stopping an animation that is running continuously involves selecting ‘Stop’ in the pop-up menu resulting from again right clicking on the graph.

We have, of course, illustrated only the simplest of ways to create an animation in MAPLE. Complete details can be found in the MAPLE manuals.

7.16 Using the Notebook

Within the GUI, the window in which MAPLE commands are entered has full capabilities as a notebook, which means in particular that it can contain textual sections, e.g., documentation and motivation, as well as executable MAPLE commands. In this window, the MAPLE prompt is \( \gg \). Notebooks are constructed out of execution groups, which may contain MAPLE statements (and the associated output)—the only groups we have so far seen—or may contain text. For purposes of organizing larger documents, these execution groups can themselves be assembled into sections and subsections, much in the way a conventional book would be assembled.

Suppose we already have on-screen a MAPLE notebook containing a few MAPLE statements and their output. Suppose, further, that we want to precede each statement with an explanatory paragraph. We could achieve that end for one of the statements by

1. Placing the cursor at the beginning of the selected statement;
2. Selecting ‘Paragraph’ from the INSERT menu and, in the resulting pop-up menu, selecting ‘Before Cursor’\(^{56}\) and finally

\(^{55}\)If the files have not in your computing environment been placed where \LaTeX can find them automatically, you will need to copy them from the directory $\$MAPLEHEAD/etc into the directory from which you are running \LaTeX before you will be able to process a MAPLE-generated \LaTeX source file.

\(^{56}\)Space will be opened above the selected statement and set to expect text.
3. Placing the cursor in the newly created space and typing the desired text, using the various pull-down menus just under the toolbar to specify font, type size, and type style; the FORMAT menu to specify alignment; . . . .

This process can, of course, be repeated to provide a textual description for each of the remaining statements. Similarly, new execution groups can be created by selecting ‘Execution Group’ from the INSERT menu.\footnote{In earlier versions of MAPLE, one inserted an execution group and then toggled it to ‘text’ by clicking ML on the button labeled with a large ‘T’ in the toolbar. That icon has been given a different function in the newer versions of MAPLE.}

In essence, the previous paragraph provides all we need to know to add documentation as an intrinsic part of a notebook. As our notebooks grow in size, however, we may wish to organize their contents into appropriate structural units and provide titles. To those ends, we need to be aware of some additional features of the notebook interface. For example, to aggregate several execution groups into a single larger group (or section), we can highlight the execution groups to be aggregated and select ‘Indent’ from the FORMAT menu. The result is the creation of a higher level group that is topped with a square icon containing a minus sign or, in more recent versions of MAPLE, with a filled, downward pointing triangle. Placing the cursor to the right of that icon, we can type a title—MAPLE will automatically use a larger and bolder type face—for the aggregation and then, after typing (RETURN), we can type—MAPLE will automatically return to the default type face—a paragraph of descriptive text to be associated with the entire aggregation. The icon heading the aggregation is active, in the sense that clicking ML repeatedly on the icon will toggle MAPLE between displaying the text and statements (and output) in the section and hiding that information. Note that, when information is hidden, the icon contains a plus sign rather than a minus sign—or points to the right rather than down. Further, note that these structures can be nested to create a document with several levels of expandability.

To add a title to the entire document, we insert an execution group at the very beginning of the document, toggle that group to ‘text’ (See footnote 58.), and type in the text of the title. Then, after highlighting that title, we adjust the font, size and positioning as appropriate.

MAPLE output in this notebook can, of course, always be quickly regenerated by executing the statements. Frequently, we may wish to suppress the presence of that output in the actual notebook. We can

- remove selected items of output by selecting the items to be removed and then selecting ‘From Selection’ from the REMOVE OUTPUT menu in the EDIT menu.
- remove all output from the notebook by selecting ‘From Worksheet’ from the REMOVE OUTPUT menu in the EDIT menu.
- hide all output by selecting ‘Hide Output’ from the HIDE CONTENT menu in the VIEW menu or, in more recent versions of MAPLE, unchecking ‘Output’ from the window that pops up on selecting ‘Show/Hide Contents . . .’ item in the VIEW menu.

Finally, we note that, after all the work of creating a notebook, the notebook itself can be saved by selecting ‘Save’ from the FILE menu and, in the resulting pop-up window, specifying the file name (default file type .mw or .mws) and the directory and clicking ML on the ‘Save’ or ‘OK’ button. In a later session, the worksheet can be loaded into MAPLE by selecting ‘Open’ from the FILE menu and identifying the file to be opened in the pop-up browser.

This short section can, of course, not do full justice to the wide spectrum of features of MAPLE’s notebook interface. You must work with it, explore the menus in the GUI, and read the manual.\footnote{Note that an execution group can be converted to a text group by placing the cursor at the beginning of that group and selecting ‘Plain Text’ from the CONVERT TO menu in the FORMAT menu. The default text color may remain red, but that can be changed with the pallette available from the ‘text color’ icon near the right end of the lower toolbar in MAPLE16.}
7.17 Miscellaneous Occasionally Useful Tidbits

7.17.1 Specifying Directories

Different operating systems use different forms to specify the path that identifies a particular directory, for example, to set the default directory or to identify the location of a file to be read or written. The character separating directories in a path depends on the operating system. In UNIX, that character is a forward slash; in Windows, it is a backslash, though a single forward slash will be properly interpreted even in Windows. Indeed, if you wish to use backslashes, each must be *doubled* because a single backslash will be flagged as an error.

7.17.2 MAPLE’s Default Directory

The default directory or current working directory defined when MAPLE starts depends on a variety of factors. In some cases, it will be the directory from which MAPLE is launched. In the GUI, the current default directory is also displayed in one of the items along the bottom of the GUI. In all interfaces, the statement

```maple
currentdir();
```

will report the directory at the time the statement is executed. Changing the directory can be accomplished in several ways, specifically

- In the GUI, clicking ML on the default directory along the bottom of the GUI will bring up a browser in which one can select a different directory.

- The statement

  ```maple
currentdir( "FullPathToNewDirectory" )
```

  will change the current directory to the specified directory.

- The statement

  ```maple
currentdir( "DirectoryName" ) or
currentdir( "/DirectoryName"
```

  will change the directory to the subdirectory *DirectoryName* of the current default directory.

- The statement

  ```maple
currentdir( ".." )
```

  will change the directory to the parent of the current default directory.

- The statement

  ```maple
currentdir( "/DirectoryName"
```

  will change the directory to the subdirectory *DirectoryName* of the parent of the current default directory.

Note the following:

---

59 See the *Local Guide* for information relating to your site.
7.17. MISCELLANEOUS OCCASIONALLY USEFUL TIDBITS

- The symbols . and .. refer to the current directory and its parent, respectively.
- When currentdir is invoked with an argument, its output is the original default directory, not the new default directory. A subsequent invocation of currentdir(); will confirm that the directory was indeed changed.

You may wish to create a user-specified initialization file (see Section 7.17.4) to assure that the (initial) default directory will be set sensibly each time you launch MAPLE.

7.17.3 MAPLE’s Search Path

Whenever an explicit path is not included in the file name of a file that MAPLE is to read, the program searches for the specified file in the sequence of directories stored in the system variable libname. The value of this variable is set by default when MAPLE is launched and can be displayed with the statement

libname;

Often, the default value of this variable will consist of something like

/apps/maple/maple Vs/lib, "."

i.e., the directory in which the MAPLE program and all of its components reside followed by the current default directory ‘.’. By default, MAPLE will search for a referenced file in the MAPLE library and then in the default directory as described in Section 7.17.2. Since libname is no more than a list of (string) components, it can be changed in the same way any other list is changed. For example, the statement

libname := ( libname, "DirectoryPath1", "DirectoryPath2", ... )

will append the specified directory paths to the existing list, the statement

libname := ( "DirectoryPath1", "DirectoryPath2", ..., libname )

will place the specified directories at the beginning of existing list, and a statement like

libname := ( "DirectoryPath1", "DirectoryPath2", ... )

will redefine the variable to include only the specified list.

7.17.4 Customizing MAPLE

When MAPLE is started up, its configuration is defined by a number of default specifications built into the program, by statements in a system-wide initialization file maintained by the system manager, and by statements in a user-created initialization file. These files are simply command files—see Section 7.11—that can contain any statement that might otherwise be typed interactively. Be aware, however, that none of the statements or their responses will be displayed as the initialization file(s) is (are) read. Even so, variables to which values are assigned will be available. As MAPLE is started, the system-wide file is executed first and then the user-created file. Thus, the

---

60While this file cannot be changed by individual users, it can be listed on the screen or printed—so users can determine what is actually contained in it.
user-created file can be used to override statements in the system-wide file, should that be desired. Together, these two files—if they exist at all—define the default configuration set up by MAPLE when finally the prompt appears.

The name and location of both the system-wide and the user-created initialization files depend on the platform on which MAPLE is running. Specifically, in Unix and Linux and on Macintosh computers, the system-wide file is $\texttt{MAPLEHEAD/lib/init}$ and the user-created file is $\texttt{USERHOME/.mapleinit}$; in Windows, these two files are $\texttt{MAPLEHEAD/}lib\texttt{/maple.ini}$ and $\texttt{c:/users/UserName/maple.ini}$, respectively. Translations of $\texttt{MAPLEHEAD}$ and $\texttt{USERHOME}$ for your site are in the Local Guide.

### 7.17.5 Restoring MAPLE’s Initial State

See item 6 in Section 7.1.

### 7.17.6 Space Curves

Sometimes it is desirable to view the trajectory of a particle in three-dimensional space. MAPLE has the ability to plot space curves from knowledge of the trajectory defined parametrically by the functions $x(t)$, $y(t)$, and $z(t)$ giving the coordinates of points on a three-dimensional path. With that information, the command `spacecurve`, which is included in the package `plots`, will calculate the coordinates of the points along the trajectory, project these points onto the two-dimensional screen, and connect consecutive points with lines. To illustrate this feature, consider the equations

\[
x(t) = \cos t \quad ; \quad y(t) = \sin t \quad ; \quad z(t) = \alpha t
\]

where $\alpha$ is a constant, describing the trajectory of a charged particle moving in a constant magnetic field directed along the $z$ axis. The starting point is, of course, to evaluate $x$, $y$, and $z$. Then, we invoke the command `plot3d` to plot the graph. The statements

```maple
> with(plots);
> alpha := 0.5;
> spacecurve( [cos(t), sin(t), alpha*t], t=0..30, numpoints=250,
             thickness=4, color="black", axes="frame",
             font=[Times, 16], view=0..20);
```

will produce the graph in Fig. 7.5. Here, in particular, the keyword `numpoints` defines the number of points at which the trajectory is evaluated in the interval on the parameter $t$.

### 7.18 Miscellaneous Commands

Several of the MAPLE commands introduced in this chapter have broad capabilities at which we have only hinted. Other occasionally useful commands have not been mentioned at all. In this section, we provide a bit fuller discussion of a few of the commands we have already met and alert you to a few others of which you should be aware. Full details, of course, are available in the MAPLE manuals.

- **assume, additionally, about**

The commands `assume`, `additionally`, and `about` attach properties to symbolic variables and report properties already attached. For example, the statements\(^{61}\)

\[^{61}\text{The first two statements might alternatively be combined in the form } \texttt{assume(n > 0, n, integer)} \text{, but that route would not illustrate the use of } \texttt{additionally}.\]
Figure 7.5: Space curve of a charged particle moving in a constant magnetic field along the \( z \) axis.

\begin{verbatim}
> assume( n > 0 );
> additionally( n, integer );
> about( n );
Originally n, renamed n~:
  is assumed to be: AndProp(integer,RealRange(1,infinity))
\end{verbatim}

tell MAPLE that \( n \) is a positive, real integer and display a message reminding us of those characteristics. Note that the restricted \( n \) is flagged with an appended tilde (but see the command \texttt{interface} below). Note also that the command \texttt{assume} applied to a particular variable \texttt{replaces} all previous assumptions applying to that variable (which explains the need for the command \texttt{additionally}). A detailed list of properties accepted by \texttt{assume} is included in the on-line help message produced with the statement \texttt{?assume}. Among those properties are \texttt{positive}, \texttt{negative}, \texttt{real}, \texttt{RealRange(a,b)}, \texttt{vector}, and \texttt{SquareMatrix}. Further, constraints can be specified by inequalities, as in \texttt{assume( a > b ) and assume( b-a > 0 )}.\footnote{Strictly, assumptions can be imposed only on variables that have no assigned value. Assigning a constant value to a variable carrying assumptions will remove those assumptions. Imposing assumptions on a variable to which an expression has been assigned risks either imposing assumptions on variables in that expression or removing the assumptions on the variable in question.}

Once an assumption need no longer be attached to a variable, it can be removed altogether by assigning the variable to itself, i.e.,

\begin{verbatim}
> n := 'n';
> about( n );
n: nothing known about this object
\end{verbatim}

Note that this statement removes \texttt{all} assumptions. If only some are to be removed, we would have to remove all of them and then reinstate the ones we wished to preserve.

Remember, too, that variables to which an assumption has been attached are flagged with a suffixed tilde. That flagging can, however, be turned off by setting the interface variable \texttt{showassumed} as described in a later paragraph in this section.
• **expand**

At base, the command `expand` distributes products over sums but, as we have seen in this chapter, the command will also invoke trigonometric addition formulae to expand expressions like \( \sin(x + y) \). Further, `expand` knows how to expand a wide variety of the common (and some of the not so common) mathematical functions, e.g., all of the trigonometric and hyperbolic functions, the Bessel functions, and the factorial and exponential functions. Finally, with the general syntax

\[
\text{expand}( \text{Expression, Constraint1, Constraint2, \ldots} );
\]

we can even expand `Expression` while telling `expand` to refrain from expanding any subexpressions that match `Constraint1, Constraint2, \ldots`.

• **convert**

In general, a statement involving the command `convert` will have the form

\[
\text{convert}( \text{Expression, Form, \ldots} );
\]

where `Expression` is an expression to be converted and `Form` selects the type of conversion to be achieved. Additional arguments may or may not be present—depending on the form specified. In Section 7.7.5, we mentioned the forms `set`, `vector`, and `listlist`; in Section 7.8, we encountered the forms `parfrac`, `trig`, and `exp`. The help message `?convert` in MAPLE16 lists over 150 possible forms, and you are urged to examine that message. Your attention is called particularly to the forms `float` for converting expressions to floating point form; `hex, binary, octal, and base` for converting numbers to equivalent expressions in other bases; `degrees` and `radians` for converting angles between these “units”; `polar, expsincos, ln, expln, sincos, and tan` for recasting expressions involving trigonometric, exponential, and logarithmic functions; and `string` for converting expressions into strings. Information about any particular form is contained in the message displayed in response to the statement `?convert, Form`.

• **simplify**

A statement using the command `simplify` can be cast in any of the forms

\[
\text{simplify}( \text{Expression} );
\]
\[
\text{simplify}( \text{Expression, Name1, Name2, \ldots} );
\]
\[
\text{simplify}( \text{Expression, assume=Property} );
\]

(among others). With the first form, `simplify` applies a long list of simplification rules (see the on-line message displayed with the statement `?simplify`) one at a time. With the second form, `simplify` applies only the rules specified by `Name1, Name2, \ldots`, (e.g., `ln, polar, power, radical, sqrt, trig`) one after the other. Finally, with the third form, `simplify` assumes that all undefined symbols in `Expression` have the specified property (`real, positive, integer, even, odd, \ldots`).

• **map**

The command `map` instructs MAPLE to apply a specified function independently to each operand of a specified MAPLE object. For example, the statements
define a matrix and then effect an element-by-element differentiation of that matrix. In this structure, the first argument to map is the function to be applied to each element in the expression identified by (or supplied in) the second argument. Any arguments beyond the second are understood to be arguments to the function identified in the first argument.

• unapply

Among other features, the command unapply facilitates the conversion of an expression into a function. For example, the statements

> eqn1 := a*x*e^(-b*x^2);

\[ eqn1 := a e^{-bx^2} \]

> eq := unapply( eqn1, x );

\[ eq := x \rightarrow a e^{-bx^2} \]

define an expression and then generate from it a function.

• interface

The command interface provides access to the parameters that control the nature of the interaction between the display device and the underlying program. Among the more useful actions are

interface( echo );  
Display current value of echo.

interface( echo=2 );  
Set so command lines are displayed when command file is read.

interface( prompt=">>" );  
Change MAPLE prompt to >>.

interface( showassumed=0 );  
Turn off explicit flagging of variables subject to assumptions.

interface( imaginaryunit=j );  
Change imaginary unit \( I \) to \( j \).

A full description will be displayed in response to the command ?interface typed at the MAPLE prompt. Note, in particular, that some interface variables set away from their default values in a particular session will not be reset to those default values when the command restart; is executed while others will be reset. Information about this issue can be found in the help page accessed by the command ?restart.

• print

The command print(\( f \)) is provided to facilitate the display of the coding invoked by functions and procedures that are defined by MAPLE command files. For example, to obtain a listing of the coding that implements the procedure DotProduct defined in the MAPLE package LinearAlgebra, we first set the interface parameter verboseproc away from its default value 1 with the statement
interface( verboseproc = 2 ): 

so that the full—rather than an abbreviated—listing of procedures will be displayed. Then, we request the listing itself with the statement

print( LinearAlgebra[DotProduct] );

Do take the time to execute these two statements and examine the resulting display.

• piecewise

Occasionally, the need to define a function that is piecewise continuous arises. MAPLE’s function `piecewise` is designed to meet that need. For example, a rectangular barrier defined by

\[
    f(x) = \begin{cases} 
        0 & x <-a \\
        1 & -a \leq x \leq a \\
        0 & a < x 
    \end{cases}
\]

could be defined and plotted in MAPLE with the statements

\[
    > f := x -> \text{piecewise}( x < -a, 0, -a <= x \text{ and } x <= a, 1, a < x, 0 );
    > a := 1;
    > \text{plot}( f(t), t=-3.0..3.0 );
\]

Here, each pair of arguments provides a condition followed by a value to be assumed by the function when the condition is satisfied. Actually, this definition can be substantially shortened to

\[
    > f := x -> \text{piecewise}( -a < x \text{ and } x < a, 1 );
\]

because the function is designed to assign the value 0 to any region of the independent variable omitted altogether from the conditions in the arguments.

• Other Commands

We mention a few additional commands briefly. Full information for each can be obtained from the MAPLE manuals.

• `series`, which produces a series more general than a Taylor series, a series that may include negative powers of the expansion variable. The function \( \sqrt{1 - x^2}/x^2 \) does not have a Taylor series about \( x = 0 \) but does have the more general series produced by `series`.

• `alias`, which allows the definition of a shorthand alias for longer commands or symbols, e.g., the statement `alias(rad=sqrt)` would allow us, if we wished, to replace the reserved word `sqrt` with the symbol `rad`. The alias is removed with the statement `alias(rad=rad)`.

• `sum()`, which sums given values over a specified range of indices. The inert form `Sum`, which does not evaluate the sum immediately, also exists.
7.19 References

In this chapter, we have introduced only the most important features of MAPLE. A full description of all commands and features is contained in the MAPLE manuals, including

- several documents and resources as described in Section 7.2.
- a number of documents and books, links to which can be found by entering “MAPLE documentation” in the search window of your browser. In particular, you should find links to
  - the MapleSoft Documentation Center at
    \[ \text{http://www.maplesoft.com/documentation_center/} \]
    which brings up a page containing links to installation instructions, the Maplesoft Online Help System, and a list of many available books using MAPLE in a variety of disciplines and for a variety of specific purposes, and
  - many other official and unofficial documents.
- a number of books and on-line materials describing a wide spectrum of uses of MAPLE in a variety of disciplines and for a variety of specific purposes. In addition to the list provided by Maplesoft, you can find books by searching on-line commercial book stores (Amazon, Barnes and Noble, ...) for, among probably many others, one or more of the phrases ‘MAPLE Computer Algebra System’, ‘MAPLE and Physics’, ‘Maplesoft’, etc.
- more specific documents on particular MAPLE commands, links to which can be found by entering ‘CommandName in MAPLE’, e.g., ‘int in MAPLE’ in the search window of your browser.

The user is urged to browse in these materials, being alert not only to the available functions but also to the assorted option variables and keywords that modify their behavior.

While on-line documentation has largely superseded printed documentation in the last decade, several now largely outdated but still potentially valuable printed materials can be found. Specifically, your local library may include one or more copies of

- the MAPLE Getting Started Guide,
- the MAPLE Learning Guide,
- the MAPLE User Manual,
- the MAPLE Programming Guide,
- the MAPLE Introductory Programming Guide, and
- the MAPLE Advanced Programming Guide.

Some of these previous manuals may be accessed at and/or downloaded from the Waterloo Maple website at \text{www.maplesoft.com}. Two originally hard-bound books, specifically


date to MAPLEV (which has long since been replaced by more recent versions). Still, these older references contain applicable though dated information—and, in particular, contain information that may not have been reprinted in the more recent printed documentation.
7.20 Exercises

7.1. Read the MAPLE manuals to find out how the commands \texttt{expand}, \texttt{factor}, and \texttt{simplify} differ from one another and then invent several examples that will reveal those differences. To help you get started, you might find it useful to compare the effect of each of these commands on the expressions

\begin{align*}
\text{(a)} & \quad \frac{x}{a+b} + \frac{y}{a+b} & \quad \text{(b)} & \quad (a+b)(c+d) + (a+b)c \\
\text{(c)} & \quad (x+y)(z+w) & \quad \text{(d)} & \quad a(b+c(d+e)) \\
\text{(e)} & \quad \frac{x}{a+b} + \frac{y}{c+d} & \quad \text{(f)} & \quad \frac{(a-x)^2}{(a^2-2ax+x^2)^3/2}
\end{align*}

but you will probably have to invent other examples as well to reveal as many of the differences as you can. Write two or three paragraphs in which you describe the action of each of these commands and the differences among them. Creating a table showing the results of applying each of the commands to each expression you select might be a useful support for your comparison.

7.2. Use MAPLE to convert each of the expressions in the left-hand column in the table below into the expression in the associated right-hand column:

\begin{align*}
\text{(a)} & \quad \frac{(a-x)^2}{(a^2-2ax+x^2)^{3/2}} \quad \Rightarrow \quad \frac{1}{|x-a|} \\
\text{(b)} & \quad \sinh(\ln(x+\sqrt{x^2+a^2}) - \ln(a)) \quad \Rightarrow \quad \frac{x}{a} \\
\text{(c)} & \quad \frac{1}{x+\sqrt{y}} \quad \Rightarrow \quad \frac{x-\sqrt{y}}{x^2-y}
\end{align*}

These samples are chosen to illustrate particularly the use of \texttt{simplify} and \texttt{expand} but other commands will surely also be needed. Write two or three paragraphs in which you describe your efforts, including some indication of approaches that were not successful. Don’t be overly concerned about the order of terms within various sets of parentheses; that order is particularly difficult to control. Focus instead on creating the general form of each desired result.

7.3. Use MAPLE to convert each of the expressions in the left-hand column in the table below into the expression in the associated right-hand column:

\begin{align*}
\text{(a)} & \quad \frac{d}{dx} \left( x^2 e^{-x^2} \right) \quad \Rightarrow \quad -2x(x^2-1)e^{-x^2} \\
\text{(b)} & \quad \sin\left(\sqrt{a^2(a+3x) + x^2(3a+x) + y}\right) \quad \Rightarrow \quad \sin((a+x)^{3/2} + y) \\
\text{(c)} & \quad cg + cf + b^2 d + 2abd + a^2 d + b c + 2abc + a^2 c \quad \Rightarrow \quad (a+b)^2(c + d) + c(f + g) \\
\text{(d)} & \quad a e^{(-b+i \omega) t} + a e^{(-b-i \omega) t} \quad \Rightarrow \quad a e^{-bt} (e^{-i\omega t} + e^{i\omega t}) \\
\text{(e)} & \quad a e^{(-b+i \omega) t} + a e^{(-b-i \omega) t} \quad \Rightarrow \quad 2a e^{-bt} \cos(\omega t) \\
\text{(f)} & \quad x^2 + y^2 + z^2 - 2a(x+y) + 2a^2 \quad \Rightarrow \quad (x-a)^2 + (y-a)^2 + z^2
\end{align*}

In several cases, you may need to invoke \texttt{op, nops}, \texttt{subs}, and/or \texttt{subsop}, but other commands will surely also be needed. Invoke microscopic dissection of the expressions only as a last resort. Write two or three paragraphs in which you describe your efforts, including some indication of approaches that were not successful. Don’t be overly concerned about the order of terms within various sets of parentheses in the final form; that order is particularly difficult to control. Focus instead on
7.20. EXERCISES

creating the general form of each desired result.

7.4. In Section 7.8.9, we set up the differential equation for a damped harmonic oscillator but then pursued the solution only for the underdamped case. Repeat the solution for the underdamped case and then find also the solutions for the critically damped and overdamped cases. Describe the differences in the physical behavior for the three cases.

7.5. The Legendre polynomials \(P_n(x)\), which are valid and useful over the interval \(-1 \leq x \leq 1\), can be defined in many ways. They emerge as the coefficients in the Taylor expansion of the generating function

\[
g(x,t) = \frac{1}{\sqrt{1 - 2tx + t^2}} = \sum_{n=0}^{\infty} P_n(x) t^n
\]

Alternatively, they can be determined from the recursion relationship

\[(2n + 1) x P_n(x) = (n + 1) P_{n+1}(x) + n P_{n-1}(x)\]

provided we include the first two \(P_0(x) = 1\) and \(P_1(x) = x\) to get started. Yet again, they can be found from application of multiple differentiation as implied by Rodrigues’ formula

\[P_n(x) = \frac{1}{2^n n!} \frac{d^n}{dx^n} \left( (x^2 - 1)^n \right)\]

However they are determined, the first half dozen of these polynomials will turn out to be

\[
P_0(x) = 1 \quad P_1(x) = x \quad P_2(x) = \frac{1}{2}(3x^2 - 1) \quad P_3(x) = \frac{1}{8}(5x^3 - 3x) \quad P_4(x) = \frac{1}{5}(35x^4 - 30x^2 + 3) \quad P_5(x) = \frac{1}{8}(63x^5 - 70x^3 + 15x)
\]

a. Use the generating function and MAPLE’s capabilities for evaluating Taylor series to find the first half-dozen Legendre polynomials, extracting each as an expression bound to a variable.

b. Start by binding the value 1 to \(P[0]\) and the value \(x\) to \(P[1]\). Then, using the recursion relationship, find the next several Legendre polynomials. Hint: The MAPLE statements might be

\[> P[0] := 1; P[1] := x;\]

and then

\[> PP := (2*n-1)*x*P[n-1]/n - (n-1)*P[n-2]/n\]

(Verify the expression on the right by using MAPLE to deduce this relationship from the standard form—the second equation in this exercise.) With these statements, you have set up \(P_0(x)\) and \(P_1(x)\) to start the recursion and then you have defined an expression \(PP\) involving \(n\) that can be evaluated at any \(n\). Once \(P_0\) and \(P_1\) have been defined, you can find \(P_2\) and then \(P_3\) and then ... with statements like

\[> P[2] := \text{eval}(PP, n=2);\]
\[> P[3] := \text{eval}(PP, n=3);\]
... 

c. Find the first half-dozen Legendre polynomials by using the command \text{diff} to evaluate Rodrigues’ formula. Hint: You might find that using a loop would simplify your approach.

d. Be clever and, using either matrices or loops constructed in MAPLE, find the values of all of the integrals

\[
\int_{-1}^{1} P_n(x) P_m(x) \, dx
\]

where \(n\) and \(m\) take on independently the values 0, 1, 2, 3, 4, 5. (There are 36 integrals to be evaluated. Try to be efficient in your coding, and remember the command \text{map}, which will be useful in constructing a single statement that evaluates the integral of each element in a multi-element structure.)
e. Within MAPLE, obtain graphs of the first six Legendre polynomials over the interval $-1 \leq x \leq 1$.

f. It is known that a function $f(x)$ defined over the interval $-1 \leq x \leq 1$ can be expanded in a series of Legendre polynomials of the form $f(x) = \sum a_n P_n(x)$ where the coefficients $a_n$ are given by

$$a_n = \frac{2n + 1}{2} \int_{-1}^{1} f(x) P_n(x) \, dx$$

Find the first six coefficients for the expansion of the function $f(x) = 0$ when $-1 < x < 0$ and $f(x) = 1$ when $0 < x < 1$. Then, construct the (partial) series representing this function and obtain a graph of that approximation to compare with the graph of the original function.

g. MAPLE actually knows quite a bit about many of the important special functions of mathematical physics. In particular, the package `orthopoly` can be loaded with the command `with(orthopoly)` and is described fully in the MAPLE manuals. Take a look at that documentation and then use the function $P$ in that package to determine the first several Legendre polynomials.

7.6. Figure 7.6 shows the circuit diagram for a Wheatstone bridge. Using Kirchoff’s laws, set up the equations from which you could determine the currents in each branch of the circuit. Then, using MAPLE, (a) solve the equations symbolically, (b) find conditions under which the current in the cross branch (through resistor $R_5$) will be zero, and (c) find the effective resistance seen by the battery. (The effective resistance is defined by the ratio $V/I$, where $I$ is the current in the branch containing the battery.)

7.7. In the scattering of a quantum wave from a rectangular barrier in particular circumstances, we find that the wave function must be expressed in three pieces in the form

$$\psi(x) = \begin{cases} Ae^{ikx} + Be^{-ikx} & x < 0 \\ D \cosh(\kappa x) + F \sinh(\kappa x) & 0 < x < w \\ Ce^{ikx} & x > w \end{cases}$$

where $k$ and $\kappa$ are constants related to the energy of the particle and the height of the barrier, $A$ is a constant reflecting the intensity of the incident beam, and $B, D, F, C$ are constants to be determined by imposing the requirement that the wave function and its first derivative be continuous both at $x = 0$ and at $x = w$, i.e., that

$$\psi(0^-) = \psi(0^+) \quad ; \quad \psi(w^-) = \psi(w^+) \quad ; \quad \left. \frac{d\psi}{dx} \right|_{0^-} = \left. \frac{d\psi}{dx} \right|_{0^+} \quad ; \quad \left. \frac{d\psi}{dx} \right|_{w^-} = \left. \frac{d\psi}{dx} \right|_{w^+}$$

where superscript plus and minus signs identify points slightly below and slightly above the indicated value of $x$, respectively. (While $k$ and $\kappa$ can be taken to be real for this barrier, the constants $A, B, C, D, and F$ may—and probably will—be complex.) Use MAPLE’s abilities to manipulate expressions to
(a) Obtain the equations determining $B$, $C$, $D$, and $F$ by imposing the stated boundary conditions on these solutions.
(b) Solve those equations for those constants (expressing each as a multiple of the constant $A$).
(c) Show that the reflection and transmission coefficients $R$ and $T$ defined by $R = |B/A|^2$ and $T = |C/A|^2$ are given by
\[ R = \left| \frac{B}{A} \right|^2 = \frac{(k^2 + k^2)^2 \sinh^2 \kappa w}{4k^2k^2 + (k^2 + k^2)^2 \sin^2 \kappa w} \quad ; \quad T = \left| \frac{C}{A} \right|^2 = \frac{4k^2 \kappa^2}{4k^2k^2 + (k^2 + k^2)^2 \sin^2 \kappa w} \]
Here, the vertical bars symbolize the absolute value of the complex number enclosed by them.
(d) Verify that $R + T = 1$.

7.8. The distribution of wavelengths $\lambda$ in the blackbody spectrum at (absolute) temperature $T$ is given by
\[ u(\lambda, T) = \frac{8\pi c h}{\lambda^5} \frac{1}{e^{h\lambda/kT} - 1} \]
where $c$ is the speed of light, $h$ is Planck’s constant, and $k$ is Boltzmann’s constant. In terms of the variable $y = ch/\lambda kT$, this function has the alternative expression
\[ \frac{(ch)^4 u(\lambda, T)}{8\pi(kT)^5} = f(y) = \frac{y^5}{e^y - 1} \]
(a) Verify this transformed form and then, using MAPLE, (b) obtain a graph of $f(y)$ versus $y$, making sure to extend the graph over an interval that includes its peak and estimate the value of $y$ at which that peak occurs; (c) show that the peak occurs for values of $y$ satisfying
\[ (y - 5)e^y + 5 = 0 \]
(d) obtain a graph of this function versus $y$; (e) make another estimate of the value of $y$ at which the original function has its maximum; and (f) show that the wavelength $\lambda_m$ at which this maximum occurs satisfies
\[ \lambda_m T = 0.28978 \times 10^{-2} \text{ m K} \]
Hints: (1) Remember that maxima in a function occur where the derivative of that function with respect to the appropriate variable is zero. (2) Because $e^y$ varies rapidly with $y$, you may have to play a bit to find a suitable range of values of $y$ over which to plot these graphs.

7.9. Use MAPLE to verify (a) that the solution
\[ x(t) = x_0 e^{-b t/(2m)} \left( \frac{b \sin \left( \sqrt{4km - b^2} t/(2m) \right)}{\sqrt{4km - b^2}} + \cos \left( \sqrt{4km - b^2} t/(2m) \right) \right) \]
presented in Section 7.8.9 satisfies the desired differential equation and the initial conditions presented at the beginning of that section and (b) that this solution and the solution
\[ y(\tau) = y_0 \sqrt{\frac{\beta}{4 - \beta^2}} e^{-\beta \tau/2} \sin \frac{\tau \sqrt{4 - \beta^2}}{2} + y_0 e^{-\beta \tau/2} \cos \frac{\tau \sqrt{4 - \beta^2}}{2} \]
obtained at the very end of the section agree with each other.

7.10. The Laguerre polynomials $L_n(x)$ can be defined in many ways. We might, for example, set $L_0(x) = 1$ and then generate each new polynomial in turn by requiring that $L_n(x)$ for $n > 0$ be a polynomial of order $n$ that is orthogonal to all previous polynomials, i.e., that
\[ \int_0^\infty e^{-x} L_n(x) L_m(x) \, dx = 0 \quad , \quad \text{all } m < n \]
and that the arbitrary overall sign remaining be resolved by requiring in addition that $L_n(0) = 1$. Alternatively, they can be determined from the recursion relationship
\[ (n + 1)L_{n+1}(x) = (2n + 1 - x)L_n(x) - nL_{n-1}(x) \]
provided we include the first two \( L_0(x) = 1 \) and \( L_1(x) = 1 - x \) to get the recursion started. Yet again, they can be found from the application of multiple differentiation as implied by Rodrigues’ formula
\[
L_n(x) = \frac{e^x}{n!} \frac{d^n}{dx^n} (x^n e^{-x})
\]
However they are determined, the first half dozen of these polynomials will turn out to be
\[
\begin{align*}
L_0(x) &= 1 \\
L_1(x) &= 1 - x \\
L_2(x) &= 1 - 2x + \frac{1}{2} x^2
\end{align*}
\]
\[
\begin{align*}
L_3(x) &= 1 - 3x + \frac{3}{2} x^2 - \frac{1}{3} x^3 \\
L_4(x) &= 1 - 4x + 3x^2 - \frac{3}{3} x^3 + \frac{1}{24} x^4 \\
L_5(x) &= 1 - 5x + 5x^2 - \frac{5}{4} x^3 + \frac{5}{24} x^4 - \frac{1}{120} x^5
\end{align*}
\]
a. Determine the first half dozen Laguerre polynomials by constructing them one at a time to satisfy the requirements of orthogonality and normalization.
b. Start by binding the value 1 to \( L[0] \) and the value \( 1 - x \) to \( L[1] \). Then, using the recursion relationship, find the next several Laguerre polynomials. *Hint:* The MAPLE statements might be
\[
> L[0] := 1; L[1] := 1-x;
\]
and then
\[
LL := (2*n-1-x)*L[n-1]/n - (n-1)*L[n-2]/n;
\]
(Verify the expression on the right.) With these statements, you have set up \( L_0(x) \) and \( L_1(x) \) to start the recursion and then you have defined an expression \( LL \) involving \( n \) that can be evaluated at any \( n \). Once \( L_0 \) and \( L_1 \) have been defined, you can find \( L_2 \) and then \( L_3 \) and then ... with statements like
\[
> L[2] := \text{eval}( LL, n=2 ) ; \\
> L[3] := \text{eval}( LL, n=3 ) ; \\
\]
...c. Find the first half-dozen Laguerre polynomials by using the command \texttt{diff} to evaluate Rodrigues’ formula.
d. Be clever and, using either matrices or loops constructed in MAPLE, find the values of all of the integrals
\[
\int_0^\infty e^{-x} L_n(x) L_m(x) \, dx
\]
where \( n \) and \( m \) take on independently the values 0, 1, 2, 3, 4, 5. (There are 36 integrals to be evaluated. Try to be efficient in your coding, and remember the command \texttt{map}, which will be useful in constructing a single statement that evaluates the integral of each element in a multi-element structure.)
e. MAPLE actually knows quite a bit about many of the important special functions of mathematical physics. In particular, the package \texttt{orthopoly} can be loaded with the command \texttt{with(orthopoly)} and is described fully in the MAPLE manuals. Take a look at that documentation and then use the function \texttt{L} in that package to determine the first several Laguerre polynomials.

7.11. Find the eigenvalues and eigenvectors of the matrix
\[
\begin{pmatrix}
0 & 1 & 0 \\
1 & 0 & 1 \\
0 & 1 & 0
\end{pmatrix}
\]

7.12. (a) Find the eigenvalues and eigenvectors of the matrix
\[
\begin{pmatrix}
2 & -1 & 0 & 0 \\
-1 & 2 & -1 & 0 \\
0 & -1 & 2 & -1 \\
0 & 0 & -1 & 2
\end{pmatrix}
\]
7.13. When a (weak) constant external electric field of magnitude $F$—we reserve $E$ for energy in this exercise—is imposed on a hydrogen atom, the energy of the states with principal quantum number $n$ shift from the energy given by the Bohr model by amounts determined by the eigenvalues of the matrix whose elements are $\langle nlm|eF|n'l'm'\rangle$, where $l$, $m$, $l'$, and $m'$ range over all possible values of those quantum numbers allowed by the particular value of $n$. If the states by which the rows and columns are labeled are ordered $|2,0,0\rangle$, $|2,1,-1\rangle$, $|2,1,0\rangle$, and $|2,1,1\rangle$, then the matrix for the state $n=2$ is

$$3e a_0 F \begin{pmatrix} 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

where $e$ is the magnitude of the charge on the electron and $a_0$ is the Bohr radius. Similarly, if the states by which the rows and columns are labeled are ordered $|3,2,2\rangle$, $|3,1,1\rangle$, $|3,2,1\rangle$, $|3,0,0\rangle$, $|3,1,0\rangle$, $|3,2,0\rangle$, $|3,1,-1\rangle$, $|3,2,-1\rangle$, and $|3,2,-2\rangle$, then the matrix for the state $n=3$ is

$$3e a_0 F \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -9/2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -9/2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -3\sqrt{6} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -3\sqrt{6} & 0 & -9/\sqrt{3} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -9/\sqrt{3} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -9/2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -9/2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Find the eigenvalues and eigenvectors of these matrices. The eigenvalues give the energy shifts for the Stark effect for $n=2$ and $n=3$ and the eigenvectors give the linear combinations of the base states (i.e., the states in the absence of the external field) out of which the states in the presence of the field emerge as the field is turned on. **Hint:** The simplest way to create a sparse matrix is to create a matrix that is all zeroes with statements like

```plaintext
mat := array(1..4, 1..4);
for i from 1 to 4 do for j from 1 to 4 do mat[i, j] := 0; end do: end do:
```

and then set the nonzero entries with statements like

```plaintext
mat[1,3] := desired value
```

7.14. As shown in Fig. 7.7, an object of mass $m$ is connected to the center of each of the four sides of a
square of sides $2\ell$ with a spring of constant $k$ and moves on a horizontal frictionless surface in the plane defined by the square. Take the equilibrium position of the object at the center of the square to be the origin of an $xy$ coordinate system, and let the springs have unstretched length $\ell_0$. For this situation, the kinetic and potential energies $KE$ and $PE$ of the object are given by

$$KE = \frac{1}{2} m_1 \dot{x}_1^2 + \frac{1}{2} m_2 \dot{x}_2^2 + \frac{1}{2} m_1 \dot{y}_1^2 + \frac{1}{2} m_2 \dot{y}_2^2; \quad PE = \frac{1}{2} k \left[ \sqrt{(\ell - x)^2 + y^2 - \ell_0} \right] + \ldots$$

where $PE$ will have three additional terms, one for each of the remaining springs. (The term shown applies to the spring connected to the right side of the square.) By definition, the Lagrangian function for this problem is given by $L = KE - PE$ and the equations of motion can be extracted from the Lagrangian by evaluating the expression

$$\frac{d}{dt} \left( \frac{\partial L}{\partial \dot{q}} \right) - \frac{\partial L}{\partial q} = 0$$

where $q$ stands first for $x$ and then for $y$. Write out the full potential energy and then use MAPLE to deduce the equations of motion $m \ddot{x} = \ldots$ and $m \ddot{y} = \ldots$ for this system. The results are quite complicated. To simplify the problem, find in detail the equations only for the cases (a) $\ell_0 = 0$, i.e., the springs have unstretched length of 0—admittedly unrealistic springs, and (b) $x$ and $y$ remain small compared to $\ell$ throughout the motion, i.e., $x/\ell \ll 1$ and $y/\ell \ll 1$ for all time. Note: Statements like `diff(L, diff(x(t),t)` will generate an error message in MAPLE. To obtain the necessary derivatives, you will have to set up the Lagrangian first with a simple variable. If, for example, you sought the equations of motion for a 1D simple harmonic oscillator, for which the Lagrangian is

$$L := (1/2)*m*xdot^2 - (1/2)*k*x^2;$$

you would first use `xdot`, say, for `diff(x(t),t)`, then evaluate the derivatives with respect to `xdot`, and finally, shift to a new variable `y(t)`, say, for the displacement, with statements like

```maple
dL_dydot := eval( diff(L,xdot), {xdot=diff(y(t),t), x=y(t)} );
dL_dy := eval( diff(L,x), {xdot=diff(y(t),t), x=y(t)} );
eq := diff( dL_dydot, t) - dL_dy = 0;
```

7.15. In the Cartesian coordinate system illustrated in Fig. 7.8, the coordinates of the two objects having mass $m_1$ and $m_2$ in the compound pendulum are given by

$$x_1 = l_1 \sin \theta; \quad y_1 = -l_1 \cos \theta; \quad x_2 = l_1 \sin \theta + l_2 \sin \phi; \quad y_2 = -l_1 \cos \theta - l_2 \cos \phi$$

Taking $\theta$ and $\phi$ to be the generalized coordinates and remembering that

$$KE = \frac{1}{2} m_1 \dot{x}_1^2 + \frac{1}{2} m_2 \dot{x}_2^2 + \frac{1}{2} m_1 \dot{y}_1^2 + \frac{1}{2} m_2 \dot{y}_2^2; \quad PE = m_1 g y_1 + m_2 g y_2$$
use MAPLE to find (a) (simple) expressions for $KE$, $PE$, and $L = KE - PE$, (b) the equations of motion by evaluating the Lagrange equations

$$\frac{d}{dt} \left( \frac{\partial L}{\partial \dot{\theta}} \right) - \frac{\partial L}{\partial \theta} = 0; \quad \frac{d}{dt} \left( \frac{\partial L}{\partial \dot{\phi}} \right) - \frac{\partial L}{\partial \phi} = 0$$

and (c) the equations of motion when both $\theta$ and $\phi$ are small. Suggestion: If you run into major difficulties keeping $l_1 \neq l_2$ and $m_1 \neq m_2$, make the lengths and masses the same. Note: See the note in the previous exercise.

7.16. The complete elliptic integrals of the first and second kinds are given by

$$K(k) = \int_0^{\pi/2} \frac{d\phi}{(1 - k^2 \sin^2 \phi)^{1/2}}; \quad E(k) = \int_0^{\pi/2} (1 - k^2 \sin^2 \phi)^{1/2} d\phi$$

Use MAPLE to evaluate these integrals to $O(k^6)$ by expanding the integrands in Taylor series before evaluating the integrals.

7.17. Find the Laplace transform of each of the functions

(a) $f(t) = t^n$, (b) $f(t) = t e^{-at}$, (c) $f(t) = \cosh(at)$, (d) $\frac{df(t)}{dt}$

and the inverse Laplace transform of each of the functions

(e) $\tilde{f}(s) = \frac{a + bs}{s^2 + \omega^2}$, (f) $\tilde{f}(s) = \frac{a}{(s^2 + 9\omega^2)(s^2 + 4\omega^2)(s^2 + \omega^2)}$

In (c), you may have to express the hyperbolic cosine in exponential form while in (f) you may have to help MAPLE’s `invlaplace` routine by first invoking `convert` with the keyword `parfrac` to expand the desired function in partial fractions.

7.18. 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 extreme 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, 0.729383, and 0.7258887, respectively.

7.19. Find symbolic expressions for the coefficients $a$, $b$, and $c$ that will cause the parabola $y = ax^2 + bx + c$ to pass through the three points $(x_1 = x_2 - \delta x, y_1)$, $(x_2, y_2)$, and $x_3 = x_2 + \delta x, y_3)$. Then integrate the parabola over the interval $x_1 \leq x \leq x_3$ and deduce Simpson’s rule

$$\int_{x_1}^{x_3} y(x) \, dx = \frac{\delta x}{3} \left( y_1 + 4y_2 + y_3 \right)$$

7.20. Create and test a command file that will define a procedure that will return the element-by-element product of two matrices having the same dimensions. The matrices are to be created with the command `array` (not `Array`). Try to code your procedure so that only the two matrices are supplied as input. If necessary, however, you may include the dimensions of the matrices as additional input.

7.21. The midpoint rule $M$ and Simpson’s rule $S$ for evaluating integrals numerically start with the assumptions that

$$\int_a^b f(x) \, dx \approx M = (b - a) f \left( \frac{a + b}{2} \right); \quad \int_a^b f(x) \, dx \approx S = \frac{b - a}{6} \left( f(a) + 4 f \left( \frac{a + b}{2} \right) + f(b) \right)$$

respectively. To deduce the first of these expressions, we approximate $f(x)$ over the interval $a \leq x \leq b$ with a constant equal to the value of $f(x)$ at the midpoint of the interval while deducing the second entails approximating the function with a quadratic polynomial that passes through $f(x)$ at the endpoints and the midpoint of the interval. The midpoint rule will clearly be $100\%$ accurate if $f(x)$ is a constant and Simpson’s rule will be $100\%$ accurate if $f(x)$ is a quadratic polynomial.
Use symbolic manipulation to show that, surprisingly, the midpoint rule is in fact 100% accurate for the linear function \( f(x) = mx + p \) and Simpson's rule is 100% accurate for the cubic polynomial \( f(x) = cx^3 + dx^2 + ex + g \). Optional: Try to construct geometric arguments that would provide insight into the correctness of these analytic results.

7.22. Using MAPLE, obtain graphs of the potential
\[
U(x) = -\frac{U_0 a^2 (a^2 + x^2)}{8a^4 + x^4}
\]
and the associated force. Note that the expression for the potential is simpler to plot if you plot not \( U(x) \) versus \( x \) but rather \( U(x)/U_0 \) versus \( x/a \), i.e., rewrite the function in the form
\[
\frac{U(x)}{U_0} = \frac{1 + \frac{x^2}{8 + \frac{\pi}{a}}}{8 + \frac{\pi}{a}}
\]
You should be able to find a similar dimensionless version of the expression giving the force.

7.23. Using MAPLE, find complete (symbolic) solutions to each of the following problems, and use MAPLE also to verify that the solutions you obtain actually do satisfy the original ODE and initial conditions.

a. \( \frac{d^2x}{dt^2} = a \), \( x(0) = x_0 \), \( v(0) = v_0 \), where \( a \) is constant, i.e., find position as a function of time for a particle moving under the action of a constant force and launched with arbitrary initial conditions.

b. \( m \frac{d^2x}{dt^2} = -eE_0 \cos(\omega t + \theta) \), \( x(0) = x_0 \), \( v(0) = v_0 \), i.e., find position as a function of time for a charged particle moving under the action of a sinusoidal force and launched with arbitrary initial conditions.

c. \( m \frac{d^2x}{dt^2} = -mg + b \left( \frac{dx}{dt} \right)^2 \), \( x(0) = 0 \), \( v(0) = 0 \), i.e., find position as a function of time for a particle released from rest at the origin and allowed to fall freely under the action of gravity and a viscous retarding force proportional to the square of the speed.

d. the differential equations
\[
m \frac{d^2x}{dt^2} = -b \frac{dx}{dt} \quad \text{and} \quad m \frac{d^2z}{dt^2} = -mg - b \frac{dz}{dt}
\]
for the motion of a projectile moving under gravity in a viscous medium when the motion starts at the origin with the initial velocity \( v = v_0 \hat{i} + v_0 \hat{k} \).

7.24. Using the functions \texttt{Gradient}, \texttt{Laplacian}, \texttt{Divergence}, and \texttt{Curl} defined in Section 7.8.10,

a. Evaluate the gradient of (i.e., the negative of the force field associated with) each of the functions
\[
V_1(x, y, z) = \frac{1}{(x^2 + y^2 + z^2)^{3/2}} \quad V_2(x, y, z) = \frac{z}{(x^2 + y^2 + z^2)^{3/2}}
\]
\[
V_3(x, y, z) = \frac{2x^2 - x^2 - y^2}{(x^2 + y^2 + z^2)^{3/2}} \quad V_4(x, y, z) = \frac{e^{-a(x^2+y^2+z^2)^{1/2}}}{(x^2 + y^2 + z^2)^{3/2}}
\]
of the Cartesian variables \((x, y, z)\).

b. Verify that each field obtained in part (a) is conservative by showing that the curl of each is zero.

c. With \( r = x \hat{i} + y \hat{j} + z \hat{k} \), show (1) that \( \nabla \times r = 0 \) and (2) that \( \nabla \cdot r = 3 \).

d. With \( V_2(x, y, z) = x^2 + y^2 + z^2 \), show that \( \nabla^2 V_2(x, y, z) = \nabla \cdot \nabla V_2(x, y, z) = 6 \).

e. Show that \( \nabla^2 V_4(x, y, z) = 0 \). (Note: Except at \( x = y = z = 0 \).)

f. Evaluate the Laplacian of \( V_4(x, y, z) \), \( \nabla^2 V_4(x, y, z) \).
7.25. Using the functions Gradient, Laplacian, Divergence, and Curl defined in Section 7.8.10,

a. Evaluate the gradient of (i.e., the negative of the force field associated with) each of the functions

\[ V_1(r, \phi, z) = \frac{1}{(r^2 + z^2)^{1/2}} \quad V_2(r, \phi, z) = \frac{z}{(r^2 + z^2)^{3/2}} \]
\[ V_3(r, \phi, z) = \frac{2z^2 - r^2}{(r^2 + z^2)^{5/2}} \quad V_4(r, \phi, z) = \frac{e^{-a(r^2 + z^2)^{1/2}}}{(r^2 + z^2)^{1/2}} \]

of the cylindrical variables \( r \) (radial), \( \phi \) (azimuthal), and \( z \) (axial).

b. Verify that each field obtained in part (a) is conservative by showing that the curl of each is zero.

c. With \( r = r \hat{r} + z \hat{k} \), show (1) that \( \nabla \times r = 0 \) and (2) that \( \nabla \cdot r = 3 \).

d. With \( V_5(r, \phi, z) = r^2 + z^2 \), show that \( \nabla^2 V_5(r, \phi, z) = \nabla \cdot \nabla V_5(r, \phi, z) = 6 \).

e. Show that \( \nabla^2 V_1(r, \phi, z) = 0 \). (Note: Except at \( r = z = 0 \).)

f. Evaluate the Laplacian of \( V_4(r, \phi, z) \), \( \nabla^2 V_4(r, \phi, z) \).

7.26. Using the functions Gradient, Laplacian, Divergence, and Curl defined in Section 7.8.10,

a. Evaluate the gradient of (i.e., the negative of the force field associated with) each of the functions

\[ V_1(r, \theta, \phi) = \frac{1}{r} \quad V_2(r, \theta, \phi) = \frac{z}{r^3} \quad V_3(r, \theta, \phi) = \frac{r^2(3 \cos^2 \theta - 1)}{r^5} \quad V_4(r, \theta, \phi) = \frac{e^{-ar^{1/2}}}{r} \]

of the spherical coordinates \( r \) (radial), \( \theta \) (polar), and \( \phi \) (azimuthal).

b. Verify that each field obtained in part (a) is conservative by showing that the curl of each is zero.

c. With \( r = r \hat{r} \), show (1) that \( \nabla \times r = 0 \) and (2) that \( \nabla \cdot r = 3 \).

d. With \( V_5(r, \theta, \phi) = r^2 \), show that \( \nabla^2 V_5(r, \theta, \phi) = \nabla \cdot \nabla V_5(r, \phi, z) = 6 \).

e. Show that \( \nabla^2 V_1(r, \theta, \phi) = 0 \). (Note: Except at \( r = z = 0 \).)

f. Evaluate the Laplacian of \( V_4(r, \theta, \phi) \), \( \nabla^2 V_4(r, \theta, \phi) \).

7.27. Create four three-component vectors \( \mathbf{A} \), \( \mathbf{B} \), and \( \mathbf{C} \) with statements like

\[ \mathbf{A} := [A_1, A_2, A_3] : \mathbf{B} := [B_1, B_2, B_3] : \mathbf{C} := [C_1, C_2, C_3] : \mathbf{D} := [D_1, D_2, D_3] : \]

Then, using the file crossdot.mpl as described in Section 7.11, show that

a. \( \mathbf{A} \times \mathbf{B} = -\mathbf{B} \times \mathbf{A} \), or, equivalently, that \( \mathbf{A} \times \mathbf{B} + \mathbf{B} \times \mathbf{A} = 0 \).

b. \( \mathbf{A} \times (\mathbf{B} \times \mathbf{C}) = (\mathbf{A} \cdot \mathbf{C})\mathbf{B} - (\mathbf{A} \cdot \mathbf{B})\mathbf{C} \)

c. \( (\mathbf{A} \times \mathbf{B}) \times \mathbf{C} = (\mathbf{A} \cdot \mathbf{C})\mathbf{B} - (\mathbf{B} \cdot \mathbf{C})\mathbf{A} \)

d. \( \mathbf{A} \cdot (\mathbf{B} \times \mathbf{C}) = (\mathbf{A} \times \mathbf{B}) \cdot \mathbf{C} \)

e. \( (\mathbf{A} \times \mathbf{B}) \cdot (\mathbf{C} \times \mathbf{D}) = (\mathbf{A} \cdot \mathbf{C})((\mathbf{B} \cdot \mathbf{D}) - (\mathbf{A} \cdot \mathbf{D})\mathbf{B} \cdot \mathbf{C}) \)

\[ ^{63} \text{The variable } \mathbf{D} \text{ is protected in MAPLE.} \]
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\(^2\) (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.

\(^2\)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.
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 \(x\), \(position\), and \(field\) identify memory cells for floating point values, the names \(I\%\), \(count\%\), and \(lower\_limit\%\) are valid names for cells storing integers, the names \(word\$\) and \(first\_name\$\) are valid names for cells storing strings, and \(item@\) 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., \(data\), \(values\%), or \(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., \(data(1\%,1\%)\), \(values%(4\%)\), or \(names$(12\%)\). 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

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

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 \(values\%)\), 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 \(7\%\).

### 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.\(^3\) We shall use the following special symbols and words for these operations:

\(^3\)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.
• Assign a value to a named memory location:

\[
\langle \text{variable} \rangle \leftarrow \langle \text{expression determining value to be assigned} \rangle
\]

e.g.,

\[
\text{val} \leftarrow \sin(\pi \ast x)
\]
\[
\text{count}\% \leftarrow \text{count}\% + 1\%
\]
\[
\text{first}\_\text{name}\$ \leftarrow "\text{David}"
\]

etc. Any valid arithmetic expression utilizing the symbols \( \ast \) for addition (or string concatenation), \(-\) for subtraction, \( \ast \) for multiplication, \(/\) for division, and \(^\hat{}\) 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 \((\sin, \cos, \atan, \sqrt{}, \exp, \ldots)\) to facilitate scientific computation.

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

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

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

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

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, \ldots. 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.

### 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 \textbf{true} or \textbf{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, \ldots). The six \textit{simple} comparisons we might want to code and the circumstances under which the computer will judge each to be \textbf{true} are enumerated in Table 9.1.

\[\text{\footnote{We declare that integer constants shall be explicitly identified with a trailing percent sign and that string constants shall be enclosed in double quotation marks. Some languages use single quotation marks, while others will accept either (provided they are paired).}}\]
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.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Truth if</th>
</tr>
</thead>
<tbody>
<tr>
<td>A@ = B@</td>
<td>A@ is the same as B@ (A@ equals B@)</td>
</tr>
<tr>
<td>A@ &lt; B@</td>
<td>A@ occurs before B@ in some collating sequence (e.g., numerical order, alphabetical order) (A@ less than B@)</td>
</tr>
<tr>
<td>A@ &gt; B@</td>
<td>A@ occurs after B@ in that collating sequence (A@ greater than B@)</td>
</tr>
<tr>
<td>A@ &lt;&gt; B@</td>
<td>A@ differs from B@ (A@ not equal to B@)</td>
</tr>
<tr>
<td>A@ &lt;= B@</td>
<td>A@ occurs before or is the same as B@ (A@ less than or equal to B@)</td>
</tr>
<tr>
<td>A@ &gt;= B@</td>
<td>A@ occurs after or is the same as B@ (A@ greater than or equal to B@)</td>
</tr>
</tbody>
</table>

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>

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

\[ \text{NOT} \langle \text{condition} \rangle \]

which we define to be true if \langle condition \rangle is false and false if \langle condition \rangle 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 ?. Thus, for example, we might at some point code a statement like

\[ \text{EQUAL?} \leftarrow N@ = M@ \]

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

\[ \text{EQUAL?} \leftarrow N@ = M@ \]

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

5We have used \leftarrow and =. In later sections, we will identify the symbols used in other languages.
• 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

```plaintext
val ← sin(pi*x); count% ← count% + 1%; first_name$ ← "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

```plaintext
U(I%, J%) ← 0.25 * ( U(I%+1%, J%) + U(I%-1%, J%) → + 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

```plaintext
BEGIN_BLOCK
  val ← sin(pi*x)
  count% ← count% + 1%
  first_name$ ← "David"
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 structures⁷ 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.
- Two-way selection, which is expressed generically with a statement like⁸

⁶Remember that spaces not in quoted strings are ignored by the compiler.

⁷Since 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.

⁸The 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.

⁹We 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

```plaintext
CNT% ← 0% ! 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.
9.1. COMPONENTS OF A PROGRAMMING LANGUAGE

IF (condition)
    THEN (block1 of statements) ! the THEN clause
    ELSE (block2 of statements) ! the ELSE clause
END_IF

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

• Multi-way selection

CASE
    OF ⟨condition1⟩ DO (block1 of statements)
    OF ⟨condition2⟩ DO (block2 of statements)
    OF ⟨condition3⟩ DO (block3 of statements)
    ...
    OF OTHERS DO (blockO of statements)
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. The OTHERS clause may be omitted altogether if ⟨blockO of statements⟩ is null. The flow diagram in Fig. 9.1(b) is often used to convey this structure.

• Repetition

LOOP
    (block1 of statements)
    EXIT_LOOP WHEN (condition)
    (block2 of statements)
END_LOOP

Here, the statements in the body of the loop are executed repeatedly until ⟨condition⟩, which is tested at the indicated point in the loop, becomes true. Looping will continue forever unless repeated execution of the statements eventually causes ⟨condition⟩ 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

I% ← 0%
LOOP
    I% ← I% + 1%
    Y(I%) ← sin( X(I%) )
EXIT_LOOP WHEN I%=N%
END_LOOP

10As 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.

11Not all compilers adopt this strategy. Some will execute the blocks associated with every true condition in the structure.
(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

\[ Y \leftarrow \sin(X) \]

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.,

\[ \text{WHILE } \langle \text{condition} \rangle \text{ DO (block of statements) } \]

and the block is not executed at all if \( \langle \text{condition} \rangle \) is false when the loop is entered. In the second version, the test is at the end of the loop, e.g.,

\[ \text{REPEAT (block of statements) UNTIL } \langle \text{condition} \rangle \]

and the block will be executed at least once, whether \( \langle \text{condition} \rangle \) 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.,

\[ \text{FOR } \% \% \leftarrow \text{MIN}\% \text{ THRU MAX}\% \text{ DO (block of statements) } \]

When this loop is executed, the body of the loop is executed for \( \% \% \) having the value \( \text{MIN}\% \), then for \( \% \% \) having the value \( \text{MIN}\%+1\% \), then for \( \% \% \) having the value \( \text{MIN}\%+2\% \), and ..., continuing until the loop has been executed for the largest value of \( \% \% \) that does not exceed \( \text{MAX}\% \). The more sophisticated form

\[ \text{FOR } \% \% \leftarrow \text{MIN}\% \text{ THRU MAX}\% \text{ STEP INC}\% \text{ DO (block of statements) } \]

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 \( \text{MIN}\% \) equals \( \text{MAX}\% \) and not at all if \( \text{MIN}\% \) exceeds \( \text{MAX}\% \)—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
9.1. COMPONENTS OF A PROGRAMMING LANGUAGE

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.

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@:\[1]  

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

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]:

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

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:

\[
\begin{align*}
\text{NUMBER_OF_TIMES@} & \leftarrow \langle \text{desired number of executions} \rangle \\
\text{COUNTER@} & \leftarrow 0\% \\
\text{LOOP} & \\
\text{EXIT_LOOP WHEN COUNTER@} & = \text{NUMBER_OF_TIMES@} \\
\langle \text{block of statements} \rangle & \\
\text{COUNTER@} & = \text{COUNTER@} + 1\% \\
\text{END_LOOP}
\end{align*}
\]

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:

\[
\begin{align*}
\text{TIMES_REMAINING@} & \leftarrow \langle \text{desired number of executions} \rangle \\
\text{LOOP} & \\
\text{EXIT_LOOP WHEN TIMES_REMAINING@} & = 0\% \\
\langle \text{block of statements} \rangle & \\
\text{TIMES_REMAINING@} & \leftarrow \text{TIMES_REMAINING@} - 1\%
\end{align*}
\]

\[12\] Note again the use of the character ! to flag comments. See footnote 9.
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 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

\[
\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} = 0
\]  

(9.1)

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

9.3. TWO LARGER ALGORITHMS

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)

$$\frac{\partial^2 U}{\partial x^2} \approx \frac{U_{i+1,j} - 2U_{i,j} + U_{i-1,j}}{\Delta x^2} (9.2)$$

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

$$U_{i,j} \approx \frac{1}{4} \left( U_{i+1,j} + U_{i-1,j} + U_{i,j+1} + U_{i,j-1} \right) (9.3)$$

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—requires 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.\footnote{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.} 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

```plaintext
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

```plaintext
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
    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 for display and/or when a program-independent permanent record of the data is desired.
9.3. TWO LARGER ALGORITHMS

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

\[
\text{FOR } i\% \leftarrow 1\% \text{ THRU N}\% \text{ DO} \\
\text{FOR } j\% \leftarrow 1\% \text{ THRU N}\% \text{ DO} \\
\text{FOR } k\% \leftarrow 1\% \text{ THRU N}\% \text{ DO WRITE } A(i\%,j\%,k\%) \\
\]

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.

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.

\footnote{In some languages, the actual values of these indices will range from 0 to $n_\star - 1$; in other languages, the values will range from 1 to $n_\star$. (The subscript $\star$ stands for $x$, $y$, or $z$.) 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 $x_{\text{min}} \leq x \leq x_{\text{max}}$ and $y_{\text{min}} \leq y \leq y_{\text{max}}$ 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

\begin{verbatim}
DIMENSION ARRAY(n_x,n_y) ! Prepare array for values
NARR% ←− 1% ! Set number of arrays
NX% ←− n_x ! in each coordinate
NY% ←− n_y
NZ% ←− 1%
DX ←− (x_{max} - x_{min})/(n_x - 1) ! Set increments
DY ←− (y_{max} - y_{min})/(n_y - 1)
\end{verbatim}

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

\begin{verbatim}
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
END_BLOCK
\end{verbatim}

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\footnote{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.} to be used for subsequent reference to the file. Next, the first five WRITE statements send five lines of

\begin{verbatim}
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, "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
\end{verbatim}

Here, the first statement identifies the file and assigns to it a number—any number will do\footnote{Be aware that some languages reserve a few channel numbers for special "files", e.g., the keyboard or the screen.}—to be used for subsequent reference to the file. Next, the first five WRITE statements send five lines of
9.4 SOLVING LAPLACE’S EQUATION WITH FORTRAN

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

\[
\text{ATTACH FILE (filename) ON CHANNEL 1 FOR READING} \\
\text{READ FROM CHANNEL 1, Comma-separated list of variables for storage of values} \\
\vdots \\
\text{CLOSE FILE ON CHANNEL 1}
\]

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.

9.4 Solving Laplace’s Equation with FORTRAN

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

Rather than systematically laying out features of FORTRAN 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 FORTRAN 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:

- Except for characters within quoted strings, FORTRAN is insensitive to case.
- Lines with a C in the first character position are comments. Further, while it is not standard FORTRAN, most FORTRAN compilers will accept comments introduced with an exclamation point.
- Legal variable names in FORTRAN begin with a letter, which may be followed by any number of characters chosen from the letters A–Z, the numeric digits 0–9, and—in extended FORTRAN-77—the underscore _, the dollar sign $, and the period .'. Note, however, that in standard FORTRAN-77 only six characters are allowed and further that in extended FORTRAN characters beyond the thirty-second will be ignored by the compiler.
Table 9.4: A FORTRAN program to solve Laplace's equation.

PROGRAM LAPLACE

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

PARAMETER( NXDIM = 15, NYDIM = 15, MAXIT = 300)
DIMENSION U( NXDIM, NYDIM )

C ***** Initialize U(I,J); set boundary conditions *****
DO I = 1, NXDIM ! Set all elements to zero
   DO J = 1, NYDIM
      U( I, J ) = 0.0
   ENDDO
ENDDO

DO J = 1, NYDIM ! Correct values on right edge
   U( NXDIM, J ) = 100.0
ENDDO

C ***** Iterate MAXIT times *****
DO ITCNT = 1, MAXIT

DO I = 2, NXDIM-1 ! Conduct one iteration
   DO J = 2, NYDIM-1
      U( I, J ) = 0.25 * ( U(I+1,J) + U(I-1,J)
                       + U(I,J+1) + U(I,J-1) )
   ENDDO
ENDDO

ENDDO

C ***** Display solution *****
DO J = 1, NYDIM
   WRITE( UNIT = 6, FMT = 200 ) ( U(I,J), I=1,NXDIM )
ENDDO

200 FORMAT(' ', 15F7.2 )
END
• For the most part, we shall exploit FORTRAN's capability for implicit data typing, where a variable whose name begins with \( I, J, K, L, M, \) or \( N \) will be understood to represent an integer while a variable whose name begins with any other letter will be understood to represent a (single-precision) floating point number.

Extended FORTRAN-77 also admits explicit data typing (and dimensioning), where variable declarations like

\[
\begin{align*}
\text{REAL IMAP, LETTER(20)} \\
\text{DOUBLE PRECISION X, Y} \\
\text{CHARACTER A} \\
\text{INTEGER ROOT, Z(30,50)}
\end{align*}
\]

specify the indicated variables as having the indicated data type and can, as for example in the first statement, override the implied implicit type for a particular variable. Any variables not explicitly typed will be typed in accordance with the rules for implicit data typing unless the program includes the statement

\[
\text{IMPLICIT NONE}
\]

in which case the compiler will complain if it encounters a variable that has not been explicitly typed.

• Statements can begin no earlier than the seventh character position in the line and cannot extend past the 72-nd character position.\(^{18}\)

• A FORTRAN program begins with a statement \texttt{PROGRAM (ProgramName)} and concludes with the statement \texttt{END}.

• The statement beginning \texttt{PARAMETER} provides a means to assign a character string to a symbol such that subsequent use of the symbol triggers a literal replacement of the symbol with the assigned character string. These symbols are \textit{not} variables, since they cannot be assigned different values during execution of the program. The \texttt{PARAMETER} statement 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 \textit{one} place.

• FORTRAN uses the symbols \( +, -, *, /, \) and \( ** \) for addition, subtraction, multiplication, division, and exponentiation (raising to a power).

• In FORTRAN, the indices of elements of arrays are enclosed in ordinary parentheses and run upwards from 1. Thus, for example, an array dimensioned with the statement \texttt{DIMENSION U(15,15)} will have elements running from \( U(1,1) \) to \( U(15,15) \) with 15 values for each index.

• FORTRAN uses the equal sign for assignment. The operators \texttt{.gt.}, \texttt{.ge.}, \texttt{.lt.}, \texttt{.le.}, \texttt{.eq.}, and \texttt{.ne.} 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 \texttt{.and.}, \texttt{.or.}, and \texttt{.not.} symbolize \texttt{AND}, \texttt{OR}, and \texttt{NOT}, respectively.

• A FORTRAN loop is introduced with the keyword \texttt{DO} and terminated with the keyword \texttt{ENDDO}. The loops in this sample program are \textit{counted} loops, a feature conveyed by following the keyword \texttt{DO} with an \textit{integer} variable name and a range of values to be assumed by the integer. A third (optional) element in that comma-separated list specifies the increment between successive values of the index; the default increment is 1. The (current) value of the index is

\(^{18}\)This feature is a hangover from the days of punched cards, when the first six and the last 8 character positions on a card with 80 character positions were assigned other meanings.
compared with the upper limit 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. Termination
occurs as soon as the current value of the index exceeds the upper limit. Thus, the loop will
be executed once if the upper limit is initially equal to the initial index and not at all if the
upper limit is smaller than the initial index as the loop is started.

- *Any* character placed in the *sixth* character position in the line will be interpreted as meaning
that the statement on that line is to be viewed as a continuation of the statement on the
previous line.

- The one *WRITE* statement directs its output to channel 6, which is the screen (and is always
“attached”—FORTRAN uses the verb “open”—and hence need not be explicitly attached and
detached (i.e., opened and closed).

- The *format* of the output produced by the *WRITE* statement is specified in the *FORMAT*
statement that has the *label* 200 placed in the first five character positions, to which the second argument
in the first set of parentheses in the *WRITE* statement refers. This specific *FORMAT*
statement tells the compiler that it should first send to the output destination a space (the quoted
space as the first argument).\(^{19}\) Then, in the line to be output, there will be fifteen—the
15 in the designation 15F7.2—floating point numbers, each presented 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 its presentation. In
this particular problem, we know that all values should be positive and no value will be larger
than 100.\(^{20}\) 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.\(^ {21,22}\)

- The *WRITE* statement producing the output appears to be embedded in a single loop (the
loop on the index J). The argument to the *WRITE* statement in the *second set* of parentheses,
however, contains an *implied* loop on the index I. We might alternatively have written the
double loop

\[
\text{DO } J = 1, \text{ NYDIM}
\]
\[
\text{ DO } I = 1, \text{ NXDIM}
\]
\[
\text{ WRITE( UNIT = 6, FMT = 200 ) } U(I,J)
\]
\[
\text{ ENDDO}
\]
\[
\text{ ENDDO}
\]

200 FORMAT(’ ’, F7.2 )

but the result would have been to use a new line for *each* value in the array.

---

\(^{19}\)When FORTRAN was developed, output was almost always to a line printer, and the *first* character in each line
of output was *not* printed. Rather, it was interpreted by the printer as a control character—new line, double space,
skip to new page, depending on the character. The character telling the printer to start a new line was a space. Thus,
we must assure that the first character sent in each line will be a space, which will not be printed. The technique
shown here is one of the ways to achieve that objective.

\(^{20}\)Solutions 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.

\(^{21}\)If we had to allow for the possibility of values ranging downward to -100, we would need one more character
position (F8.2) to provide the position for a possible minus sign and guarantee at least one space between any
possible consecutive entries in each line.

\(^{22}\)One alternative to the *F* format is *I*, which is used for integers, e.g., 4110 for four integers each-right justified
in a field ten 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., 3E15.6
for three such numbers 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
E15.6, 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.
This program is stored in a file named `laplace.f` in the directory `$HEAD/fortran`. 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:

```bash
f77 -o laplace.xf laplace.f
./laplace.xf
```

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.

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, we could use the simple statement

```bash
./laplace.xf > laplace_f.dat
```

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.f` by replacing the loop(s) containing the `WRITE` statement with the statements

```fortran
OPEN( UNIT = 1, FILE = 'laplace_f.dat', STATUS = 'NEW' )
DO J = 1, NYDIM
   WRITE( UNIT = 1, FMT = 200 ) ( U(I,J), I=1,NXDIM )
ENDDO
CLOSE( UNIT = 1 )
200 FORMAT(' ', 15F7.2 )
```

where we have simply added the `OPEN` and `CLOSE` statements and edited the `WRITE` statement to specify the unit (channel) on which we opened the file. The program so edited is stored in the file `laplace_file.f` in the directory `$HEAD/fortran`. Once copied to the default directory, it can be compiled, linked, and run as with `laplace.f` above (see the *Local Guide*). In the end, the output file `laplace_f.dat` will be created in the default directory.

A graphical display of the resulting file could be generated by IDL by issuing to IDL the statements

```idl
IDL> openr, 1, 'laplace_f.dat'  
Open file.
IDL> soln = fltarr(15,15)     
Create array for solution.
IDL> readf, 1, soln            
Read in solution.
IDL> close, 1                  
Close file.
IDL> surface, soln             
Display solution as mesh surface.
IDL> device, decomposed=0      
Load a particular color table.
IDL> loadct, 3                 
Display solution as a shaded surface.
IDL> shade_surf,soln
```

---

23Throughout this book, we shall assume that we are using the FORTRAN-77 routines in a UNIX environment. In UNIX, `f90` and `f95` would invoke the FORTRAN-90 and FORTRAN-95, respectively. In other operating systems, the FORTRAN compiler might be invoked with the command `fortran` or `gfortran`, and the characters `./` may or may not be necessary to launch an executable program in the current directory. Further detail on how to invoke the FORTRAN compiler and run an executable program in your environment will be described in the *Local Guide*.

24Incidentally, note that, if necessary, execution of a FORTRAN program can usually be aborted by typing `CTRL/C`.

9.5 Additional Features of FORTRAN

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.

9.5.1 Multi-Way Selection

The basic decision-making structure in FORTRAN involves key words, conditions, and actions in the general pattern conveyed by the listing

```fortran
IF ⟨condition1⟩ THEN 
  ⟨block1 of statements⟩
ELSE IF ⟨condition2⟩ THEN 
  ⟨block2 of statements⟩
  ...
ELSE IF ⟨conditionn⟩ THEN 
  ⟨blockn of statements⟩
ELSE 
  ⟨catch-all block of statements⟩
END IF
```

Two-way selection is, of course, simply a version of multi-way selection in which there is only one condition in this sequence. Further, the ELSE clause can be omitted if it would contain no statements, and the keywords in this structure bracket the various blocks of statements so no additional blocking structure is necessary. The blocks can contain any number of statements, each on its own line, and the conditions can be logical variables, single logical expressions, or compound logical expressions.

If it should happen that the ELSE clause in a two-way selection is null and the THEN clause contains only a single statement, then the coding

```fortran
IF ⟨condition⟩ ⟨Single FORTRAN statement⟩
```

9.5.2 Reading Data from Keyboard

Data can be read from the keyboard in FORTRAN with a variety of statements. In the simplest approach, we might use the command `write` to print a prompting message conveying the item wanted and then the command `read` to read that item from the keyboard and store it in a suitable variable with statements like

```fortran
WRITE( *, * ) 'Suitable prompting message: '
READ( *, * ) VALUE
```

where we have allowed the compiler to choose the default input and output devices (first asterisk in each statement; unit 6 for the `write` statement; unit 5 for the `read` statement). Further, the compiler will automatically identify the prompting message as a string and the entered value as a floating point value (second asterisk in each statement). The statements
**9.5. ADDITIONAL FEATURES OF FORTRAN**

```
WRITE( UNIT=*, FMT=** ) 'Suitable prompting message: '
READ( UNIT=*, FMT=** ) VALUE
```

are equivalent, though they use keywords to make explicit the function of the two asterisks.

While it is probably wise to use “free” format on input so we don’t have to be too careful about the position of the input in the line, we might at times want to specify the format of the output more specifically, for example, with a statement like

```
WRITE( UNIT=*, FMT='(1X,A)' ) 'Suitable prompting message: '
```

or with a pair of statements like

```
WRITE( UNIT=*, FMT=157 ) 'Suitable prompting message: '
157 FORMAT( ' ', A )
```

Finally, note that two values would be entered in response to the prompting message if the `read` statement were coded like

```
READ( *, * ) VALUE, INDEX
```

The values would be separated by one or more spaces, one or more tabs, or simply a comma. In this specific example, of course, the first value would be stored as a single-precision floating point value while the second would be converted to an integer if necessary and stored as an integer.

**9.5.3 Additional Loops**

Standard FORTRAN-77 does not have any loops beyond the `do/enddo` loop already introduced. Extended versions of FORTRAN, however, may possess other structures. The loop

```
DO WHILE (condition)
  (block of statements)
ENDDO
```

which executes (block of statements) repeatedly as long as (condition) remains true is a particularly common extension.

**9.5.4 Subroutines, Functions, and Common Storage**

As with most computer languages, FORTRAN also admits the possibility of creating new commands/functions through the use subprograms. In essence, a subprogram 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. Subprograms come in two types, In outline, we define a `subroutine` with a structure like

---

25 Beyond the formats `F`, `E`, and `I` described in footnote 22, the descriptor `A` conveys to the compiler that it is to read characters from the input line. If only `A` is specified, the compiler reads and counts characters until `(RETURN)` is typed; if the format descriptor takes the form `AN`, the compiler reads `N` characters.

26 Check your local FORTRAN manual for the features of the FORTRAN compiler available at your site.
SUBROUTINE SubName( Arg1, Arg2, ... )
    Statements coding the desired action.
RETURN
END

Here, SubName is a user-assigned name for the subroutine, and Arg1, Arg2, ... are the names of the variables that communicate between the calling program and the subroutine, either by providing input to the subroutine or providing values returned to the calling program. All other variables in the subroutine are invisible to the calling program (and can, in fact, have the same names as variables in the calling program without generating conflict). The statements defining the subroutine must appear together in the listing of a full program but they can be placed either before or after the main program. Once in place, the subroutine itself can be invoked at any point in the program with the single statement

CALL SubName( Arg1, Arg2, ... )

This time, the arguments can be either variable names identifying the storage location for values to be submitted to (or returned by) the subroutine or, for quantities input to the subroutine, 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 subroutine need not be—and in general will not be—the same as those used in defining the subroutine in the first place. Data types and dimensions must match, but names need not.

The second type of subprogram is called a function and allows us to supplement the standard functions built into FORTRAN. A function is defined with a structure like

FUNCTION FuncName( Arg1, Arg2, ... )
    Statements coding the desired function.
    FuncName = ⟨value to be returned⟩
RETURN
END

Here, FuncName is the user-assigned name for the function, and the arguments can only be used for input to the function. In contrast to subroutines, which are called simply by asserting their names, user-defined functions are invoked with a statement like

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

where VarName is the name of the variable into which the function will deposit the result it returns to the invoking program.

In later chapters, we will at times find it necessary to “sneak” values of some variables into a subroutine or function without using the “official” arguments. Named common storage is designed to permit that operation. We simply invoke a directive such as

COMMON /ArbitraryName/ Var1, Var2, ...

which tells the compiler to store the values of the indicated variables in a special area of memory to which it assigns the specified name. Though only the order, the dimensions and the data types, not the actual names of the variables must be the same, similar statements must appear in each subprogram that needs access to the identified variables. In this way, any one of the subprograms containing the directive COMMON can have access to values placed in the common area by any other of those subprograms.

---

27Note, incidentally, that the slashes enclosing the name of the common area are part of the syntax; they must be there.
9.6 Solving Laplace’s Equation with C

Note: All C programs (*.c) and C-created data files (*.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 9.5. 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.
- 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.
Table 9.5: A C program to solve Laplace’s equation.

/* 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( "%.7f", U[i][j] );
            printf( "\n" );
        }
}
• 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. 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.

• The printf statement that sends a single value of \$ 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

---

28 Solutions 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.

29 If 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.

30 One 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.
for loop and the statement printf("\n"); must be blocked together to form the “object” of the outer for loop.

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:

```
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.

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

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

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

```
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

```
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.

A graphical display of the data in the resulting file could be generated by IDL by issuing to IDL the statements

---

31 Throughout 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.

32 Incidentally, note that, if necessary, execution of a C program can usually be aborted by typing ⟨CONTROL/C⟩.
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

\[
\begin{align*}
\text{if} & \langle \text{condition1} \rangle \\
& \langle \text{block1 of statements} \rangle \\
\text{else if} & \langle \text{condition2} \rangle \\
& \langle \text{block2 of statements} \rangle \\
& \ldots \\
\text{else if} & \langle \text{conditionn} \rangle \\
& \langle \text{blockn of statements} \rangle \\
\text{else} & \langle \text{catch-all block of statements} \rangle
\end{align*}
\]

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

\[
\begin{align*}
\text{switch} & \langle \text{expression} \rangle \\
& \{ \\
& \text{case} \langle \text{constant-expression1} \rangle : \langle \text{block1 of statements} \rangle \\
& \text{case} \langle \text{constant-expression2} \rangle : \langle \text{block2 of statements} \rangle \\
& \ldots \\
& \text{case} \langle \text{constant-expressionn} \rangle : \langle \text{blockn of statements} \rangle \\
& \text{default}: \langle \text{block-def of statements} \rangle \\
& \}
\end{align*}
\]

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

```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

```c
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

```c
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

```c
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

```c
return (expression giving value to be returned)
```

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

```c
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.
9.8 Solving Laplace’s Equation with IDL

The algorithm described in Section 9.3.1 is readily implemented in IDL with the statements

\[
\begin{align*}
    \text{u} & = \text{fltarr(15,15)} & \text{u}[14,*] & = 100.0 \\
    \text{for } \text{itcnt} & = 1, \text{ 30 do } $ \\
        \text{for } i & = 1, 13 \text{ do } $ \\
            \text{for } j & = 1, 13 \text{ do } $ \\
                \text{u}[i,j] & = 0.25*(\text{u}[i+1,j]+\text{u}[i-1,j]+\text{u}[i,j+1]+\text{u}[i,j-1])
\end{align*}
\]

Here, the two statements on the first line establish a $15 \times 15$ matrix of zeros and adjust the values in the final column to the stipulated boundary values. Then, the remaining statement (which contains three nested for loops and occupies four physical lines) executes 30 iterations (the outermost for loop of the double loop it contains). The innermost statement simply expresses the substance of Eq. (9.3). Note that the inner loops do not include points on the boundary of the region in which a solution is sought, so the values in the array u at its “edges” are left alone throughout the 30 iterations.

Once the solution has been completed, of course, we could display it on the screen with any number of statements, the simplest of which probably are

IDL> surface, u

or

IDL> shade_surf, u

though these could be embellished by exploiting optional arguments and a variety of keywords to rescale the axes, label the display, etc.

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 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.\footnote{Remember that indices in IDL start at 0, so a 15-element array has indices running from 0 to 14.}\footnote{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.}
9.12. CREATING AND STORING TWO-DIMENSIONAL SCALAR ARRAYS

9.12.1 ... with FORTRAN

A program in FORTRAN to create a file containing values of the two-dimensional irradiance from an illuminated square aperture begins with a “preamble” in which the program is identified and necessary variables are dimensioned. 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 (1) identify the program, (2) use the \texttt{PARAMETER} statement so that the dimensions of the array are explicitly present in only one place, facilitating changes, (3) dimension the one needed array, and (4) set appropriate values for a few constants with the lines\textsuperscript{35}

\begin{verbatim}
PROGRAM IRRAD

PARAMETER( NARR = 1, IX = 50, IY = 50, IZ = 1 )

DIMENSION RI( IX, IY ) ! For irradiance

THREEPI = 3.0*3.1415926535 ! Set constant
DX = 2.0*THREEPI/(IX-1.0) ! Set increments
DY = DX

(Here, we use RI rather than I for the array because—with implicit data typing—the latter would anticipate only integer values for the elements of the array.) Next, we evaluate the irradiance given by Eq. (9.4) with the lines\textsuperscript{36}

\begin{verbatim}
DO J = 1, IY
   YF = (J-1)*DY - THREEPI
   DO I = 1, IX
      XF = (I-1)*DX - THREEPI
      RI(I,J) = ( (SIN(XF)/XF)**2 ) * ( (SIN(YF)/YF)**2 )
   ENDDO
ENDDO
\end{verbatim}

Finally, we open the file, write the (nine) heading lines and then the array into the file, and close the file with the lines in Table 9.6. A full listing of this program will be found in Section 9.A. The program itself is stored in the file \texttt{irrad.f} in the directory \$HEAD/fortran. Once that file has been copied to the default directory, it can be compiled, linked, and run with commands as described in the \textit{Local Guide}. In UNIX, those commands might be

\begin{verbatim}
f77 -o irrad.xf irrad.f
./irrad.xf
\end{verbatim}

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

\textsuperscript{35}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.

\textsuperscript{36}Program notes: (1) With IX=50, the loop controlled by the statement \texttt{DO I = 1, IX} is executed first for \texttt{I=1}, then for \texttt{I=2}, \texttt{I=3}, ..., continuing until it has been executed for \texttt{I=50}, which is the index of the IX\textit{th} element. (2) With I and J ranging from 1 to 50 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 double asterisk ** symbolizes raising to a power, i.e., \texttt{R**T} evaluates \texttt{r^T}. 
CHAPTER 9. INTRODUCTION TO PROGRAMMING

Table 9.6: Final lines in program irrad.f. Here, we use explicit FORMAT statements for text output and “list-directed” format—symbolized by the component FMT = * (which gives the compiler the authority to choose what it thinks to be an appropriate format)—for numerical output. Note also the blank space at the beginning of each line of text. This initial character is not printed; instead, it signals the printer to start a new line.

```fortran
OPEN( UNIT = 1, FILE = 'irrad_f.dat', STATUS = 'NEW' )

WRITE( UNIT = 1, FMT = 10 )
10 FORMAT( ' Irradiance; Author: David M. Cook; Date: 15 June 1995' )

WRITE( UNIT = 1, FMT = 20 )
20 FORMAT( ' Fraunhofer diffraction at square aperture' )

WRITE( UNIT = 1, FMT = 30 )
30 FORMAT( ' Program described in CPSUP' )

WRITE( UNIT = 1, FMT = 40 )
40 FORMAT( ' **' )

WRITE( UNIT = 1, FMT = 50 )
50 FORMAT( ' **' )

WRITE( UNIT = 1, FMT = * ) NARR

WRITE( UNIT = 1, FMT = * ) IX

WRITE( UNIT = 1, FMT = * ) IY

WRITE( UNIT = 1, FMT = * ) IZ

DO J = 1, IY
  DO I = 1, IX
    WRITE( UNIT = 1, FMT = * ) RI(I, J)
  END DO
END DO

CLOSE( UNIT = 1 )
END
```

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 9.7.37 Then we set appropriate values for a few constants with the lines

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

37In the general structure of Section 9.3.2, we should here be imagining that we are preparing to store one 50 × 50 × 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.
Next, we evaluate the irradiance given by Eq. (9.4) with the lines\(^{38}\)

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

Finally, we open the file, write the (nine) heading lines and then the irradiance into the file, and close the file with the lines listed in Table 9.8 A full listing of this program will be found in Section 9.B. The program itself is stored in the file \texttt{irrad.c} in the directory \texttt{$\$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 \textit{Local Guide}. In UNIX, those commands might be

\[
\text{cc -o irrad.xc irrad.c -lm}
\]

\texttt{./irrad.xc}

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

---

\(^{38}\text{Program notes: (1) With xdim=50, the loop controlled by the statement }\text{for}(\ x=0;\ x<\text{xdim};\ x++ )\text{ is executed first for }x=0,\text{ then for }x=1,\ x=2,\ldots,\text{ continuing until it has been executed for }x=49,\text{ which is the index of the xdim}^{th}\text{ element. (2) With }x\text{ and }y\text{ ranging from 0 to 49 and dx and dy defined as }6\pi/49,\text{ xf and yf range from }-3\pi\text{ to }3\pi\text{ in 49 equal steps. (3) Note the position of the statement evaluating }yf.\text{ For computational efficiency, it has been placed in the }outer\text{ loop. (4) The function }\text{pow}(r,t)\text{ evaluates }r^t.\)
Table 9.8: 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 );
fprintf( fptr, "%d\n", zdim );

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

fclose( fptr );
```

9.12.3 ... with IDL

Note: All IDL programs (*.pro) and all IDL-created data files (*.idl.dat) in this chapter can be copied from the directory $HEAD/idl, where (as defined in the Local Guide) $HEAD must be replaced by the appropriate path for your site.

In IDL, creation of an internal array containing the values of the function of Eq. (9.4) is facilitated by the Lawrence-written IDL procedure `lugen_grid`. As described in Section 2.12.1, this procedure generates two two-dimensional arrays, one containing the $x$ coordinates of a grid of points uniformly spaced in a region of the $xy$ plane and the other containing the $y$ coordinates of points in that grid. Thus, we would create the desired array of values representing the function $I(x,y)/I_0$ with the statements\(^{39}\)

```idl
IDL> lugen_grid, x, xrange=[-3.0*pi, 3.0*pi], nx=49, $
IDL> y, yrange=[-3.0*pi, 3.0*pi], ny=49
IDL> I = ( (sin(x)/x)^2 ) * ( (sin(y)/y)^2 )
```

This array can be written to a file having the format described in Section 9.3.2 by using the Lawrence-written pro-file `lugen_list.pro`. The statement\(^{40}\)

\(^{39}\)Remember that, in IDL, array indices start at 0, so division of the interval into 49 segments will result in 50 grid points along each axis and in indices running from 0 to 49.

\(^{40}\)With indices starting at zero, the entry in the upper left corner of the array $I$ is the $(0,0)$ entry. Further—and this wrinkle is extremely important—remember that values for which the first index of a two-dimensional array in IDL stays fixed are in a column while values for which the second index stays fixed are in a row. That is, as IDL views two-dimensional arrays internally, the first index is the column index and the second index is the row index. IDL is entirely consistent in this internal interpretation, but we must be careful both in writing internal arrays out into files and—later—in reading those files into internal IDL arrays. Otherwise, we may effect an inadvertent transposition of the array. The procedure `lugen_list` pays proper attention to this wrinkle.
IDL> lugen_list, I

calls lugen_list and tells IDL to write the array I, which must already exist in the IDL workspace, to a file. When this command is entered, a window similar to that in Fig. 9.4 appears. The user enters a filename, a title, author, date, and up to four lines of comments in the various text-entry boxes. After you have typed in an item (all but the filename are optional), click the left mouse button on the next item (or press the TAB key) to move to the next item. After all items have been entered, click on the ‘SAVE’ button to complete the execution of lugen_list and write the data into the specified file in the default directory. The text-entry window will automatically close and the message

File ‘irrad_idl.dat’ saved successfully!

will appear in the IDL window. As implied by this line, the data file was here named irrad_idl.dat.

The behavior described in the previous paragraph is managed by a linked pair of pro-files. The one called lugen_list, which is listed with extensive comments in Section 9.C, creates the display illustrated in Fig. 9.4, and then uses the IDL procedure xmanager to pass control to the associated pro-file lugen_list_event, which is listed—again with comments—in Section 9.D. Xmanager simply waits for mouse clicks and then, with each such event, invokes lugen_list_event, passing to that procedure an identifier which lugen_list_event uses to decide which portion of its coding to execute in response to the event detected by xmanager. Most mouse clicks will be in text-entry boxes, and they are ignored by lugen_list_event. As conveyed in detail in the listing, clicking ML on the ‘SAVE’ button triggers the reading of the information in the various text-entry boxes, a writing of that information and the specified array into the specified file, and a return to the main IDL prompt. Clicking ML on the ‘CANCEL’ button effects an abrupt return to the IDL prompt, aborting the activity initiated by lugen_list.\footnote{The default filename is ‘untitled.dat’ and the default comments are ‘**’. The user should edit these entries as appropriate.}

\footnote{A Lawrence-written IDL procedure for reading files written by lugen_list.pro is described in Section 9.16.3.}
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

\[
p(x, y, z) = \frac{1}{2\pi(27)^{3/4}} e^{-2\rho/3} \left(\frac{3z^2}{\rho^2} - 1\right)^2 = \frac{1}{2\pi(27)^{3/4}} e^{-2\rho/3} (9z^4 - 6z^2 \rho^2 + \rho^4) \tag{9.5}
\]

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

\[
\rho = \sqrt{x^2 + y^2 + z^2} \tag{9.6}
\]

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.

9.13.1 ... with FORTRAN

A FORTRAN program for creating and storing a three-dimensional scalar array is similar to the program presented in Section 9.12.1. 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.E, is named \texttt{pdens.f} and contains sections

- Identifying the program.
- Setting parameters for the number of arrays, the dimensions of each array, the ranges of coordinates on the three axes, and the lowest coordinate on each axis.
- Dimensioning the one \( 30 \times 30 \times 30 \) array needed for the probability density and evaluating several useful constants.
- 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 center of the 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, \texttt{pdens.f} can be copied from the directory \$HEAD/fortran 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.\footnote{For instructions on how to compile and run the program, see Section 9.12.1 and the \textit{Local Guide}.} Once the program has been executed, the file \texttt{pdens.f.dat} will exist in the default directory.
9.13. CREATING AND STORING THREE-DIMENSIONAL SCALAR ARRAYS

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.F, 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 $[xrange, yrange, zrange]$ for the interval on each axis and $[x0, y0, z0]$ 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 $rhos$ 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.\footnote{For instructions on how to compile and run the program, see Section 9.12.2 and the Local Guide.} Once the program has been executed, the file `pdens.c.dat` will exist in the default directory.

9.13.3 ... with IDL

The procedure for using IDL to generate three-dimensional scalar arrays is similar to the method described in Section 9.12.3. The array representing the probability density in Eq. (9.5) is created using the IDL statements

```idl
IDL> lugen_grid, x, xrange=[-10.0, 10.0], nx=29, $  
IDL> y, yrange=[-10.0, 10.0], ny=29, $  
IDL> z, zrange=[-10.0, 10.0], nz=29  
IDL> P = fltarr(30,30,30)  
IDL> pfactor = 1.0/(2.0*!pi*27.0*27.0*27.0)  
IDL> zs = z*z  
IDL> rhos = x^2 + y^2 + zs  
IDL> rho = sqrt(rhos)  
IDL> P = pfactor * exp(-2*rho/3) * (9*zs^2 - 6*zs*rhos + rhos^2)
```

The first three lines produce a grid of values for the evaluation of the scalar field.\footnote{We here use the optional arguments to `lugen grid`. These arguments specify ranges on the $z$ coordinate. Their presence triggers the creation of three-dimensional arrays containing the $x$, $y$, and $z$ coordinates of a grid of points in three dimensions.} Here, $x$, $y$, and $z$ each ranges from $-10.0$ to $10.0$, and the interval is divided into 29 segments. The fourth line creates a three-dimensional $30 \times 30 \times 30$ array $P$. The remaining lines evaluate several intermediate quantities and then evaluate $P$. Finally, with the statement
Figure 9.5: A waveguide with rectangular cross section.

IDL> lugen_list, P

we write the data into a file in the desired format.\textsuperscript{46,47} After the filename—we here choose \texttt{pdens_idl.dat}—and optional information have been typed in the \texttt{lugen_list} window (see Fig. 9.4) and ML has been clicked on the ‘SAVE’ button, the data file will be saved in the default directory.

\section{9.14 Creating and Storing Two-Dimensional Vector Arrays}

Display of a \textit{vector} field in two dimensions requires a two-dimensional \textit{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 \textit{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. 9.5, we take the waveguide to be oriented so that the wave propagates in the positive \textit{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 \textit{z} direction and consider the particular TE waves whose electric field does not depend on \textit{z}. In mksa units, the (complex) fields in this guide are given by\textsuperscript{48}

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

\textsuperscript{46}The procedure \texttt{lugen_list} invoked here is identical for both two- and three-dimensional scalar and vector arrays and is described in Section 9.12.3.

\textsuperscript{47}A Lawrence-written IDL procedure for reading files written by \texttt{lugen_list.pro} is described in Section 9.16.3.

\textsuperscript{48}For a derivation of the magnetic field for a wave propagating in the \textit{z} direction and a discussion on wave guides in general, see \textit{The Theory of the Electromagnetic Field} by David M. Cook (Prentice-Hall, Englewood Cliffs, NJ, 1975) or \textit{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 \( \mathbf{E} \) and \( \mathbf{H} \)) specifically at \( t = 0 \), we find that
\[
\mathbf{E}(\mathbf{r}) = A \cos \kappa_x x \sin \frac{n \pi y}{b} \hat{k}
\]
\[
\mathbf{H}(\mathbf{r}) = \frac{An\pi}{\omega \mu_0 b} \sin \kappa_x x \cos \frac{n \pi y}{b} \hat{i} - \frac{A\kappa_x}{\omega \mu_0} \cos \kappa_x x \sin \frac{n \pi y}{b} \hat{j}
\]  
(9.10)

To simplify these equations further, we divide \( \mathbf{E} \) by \( A \) and \( \mathbf{H} \) by \( An\pi/\omega \mu_0 b \), and we elect to measure lengths in units of \( b \) by introducing the variables \( \bar{y} = y/b \) and \( \bar{x} = x/b \). Equation (9.10) then becomes
\[
\mathbf{E}(\mathbf{r}) = \mathbf{E}(\mathbf{r}) = A \cos \kappa_x \bar{x} \sin n \pi \bar{y} \hat{k}
\]
\[
\mathbf{H}(\mathbf{r}) = \frac{H(\mathbf{r})}{An\pi/\omega \mu_0 b} = \sin \kappa_x \bar{x} \cos n \pi \bar{y} \hat{i} - \frac{\kappa_x}{n \pi} \cos \kappa_x \bar{x} \sin n \pi \bar{y} \hat{j}
\]  
(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
\[
\begin{align*}
\overline{E}_z &= \cos \kappa_x \bar{x} \sin n \pi \bar{y} \\
\overline{H}_x &= \sin \kappa_x \bar{x} \cos n \pi \bar{y} \\
\overline{H}_y &= -\frac{\kappa_x}{n \pi} \cos \kappa_x \bar{x} \sin n \pi \bar{y}
\end{align*}
\]  
(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
\[
\begin{align*}
\overline{E}_z &= \cos 2\pi \bar{x} \sin 2\pi \bar{y} \\
\overline{H}_x &= \sin 2\pi \bar{x} \cos 2\pi \bar{y} \\
\overline{H}_y &= -\cos 2\pi \bar{x} \sin 2\pi \bar{y}
\end{align*}
\]  
(9.14)

where \( \bar{x} \) can range over any values—we choose \( 0 \leq \bar{x} \leq 1 \)—but, to be inside the guide, \( \bar{y} \) is confined to the region \( 0 \leq \bar{y} \leq 1 \). Each component of these fields can now be represented by a two-dimensional array. The \( \mathbf{H} \) field, which has two non-zero components, then is translated into two such arrays; the \( \mathbf{E} \) field, which has only one non-zero component, requires only one such array. To be explicit, we determine values of \( \overline{E}_z(\bar{x}, \bar{y}), \overline{H}_x(\bar{x}, \bar{y}), \) and \( \overline{H}_y(\bar{x}, \bar{y}) \) over the region \( 0 \leq \bar{x}, \bar{y} \leq 1 \), dividing each axis into 30 segments of length 1.0/29.0, which will entail evaluating the field components at 30 values of \( \bar{x} \) and 30 values of \( \bar{y} \)—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 \( 30 \times 30 \times 1 \) three-dimensional arrays. We shall, however, view the structure as a single \( 30 \times 30 \times 1 \times 3 \) array \( \mathbf{H} \), with \( \mathbf{H}(\ast, \ast, 1, 1) \) storing \( \overline{H}_x \), \( \mathbf{H}(\ast, \ast, 1, 2) \) storing \( \overline{H}_y \), and \( \mathbf{H}(\ast, \ast, 1, 3) \) storing \( \overline{E}_z \).49

---

49 We have supposed indices starting at 1. In languages where indices start at 0, the associations would, of course, be \( \mathbf{H}(\ast, \ast, 0, 0) \) storing \( \overline{H}_x \), \( \mathbf{H}(\ast, \ast, 0, 1) \) storing \( \overline{H}_y \), and \( \mathbf{H}(\ast, \ast, 0, 2) \) storing \( \overline{E}_z \).
9.14.1 ... with FORTRAN

A FORTRAN program for creating the necessary arrays is similar to the program discussed in Section 9.12.1. First, we identify the program, dimension necessary variables, and set various constants with the lines\(^{50,51}\)

```fortran
PROGRAM WAVEGD

PARAMETER( NARR = 3, IX = 30, IY = 30, IZ = 1 )

DIMENSION H( IX, IY, IZ, NARR ) ! For field components

TWOPI = 2.0*3.1415926535; ! Set constant
DX = TWOPI/(IX-1.0) ! Set increments
DY = DX

Then we evaluate the field components in Eq. (9.14) with the lines

```fortran
DO I = 1, IX
   XF = (I-1)*DX
   SXF = SIN(XF)
   CXF = COS(XF)
   DO J = 1, IY
      YF = (J-1)*DY
      SYF = SIN(YF)
      CYF = COS(YF)
      H(I,J,1,1) = SXF * CYF ! Hx
      H(I,J,1,2) = -CXF * SYF ! Hy
      H(I,J,1,3) = -H(I,J,1,2) ! Ez
   ENDDO
ENDDO
```

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.G. The program itself is stored in the file `wavegd.f` in the directory `$HEAD/fortran`. 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.\(^{52}\) Once the program has been executed, the file `wavegd.f.dat` will exist in the default directory.

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 9.10. the main routine continues by setting appropriate values for a few other quantities with the lines

\(^{50}\)Note that the third index of \(H\) is always one, the only value when the dimension of the array in that index is 1.

\(^{51}\)Note also that, for computational efficiency, various quantities involving only \(XF\) have been calculated in the outer loop.

\(^{52}\)For instructions on how to compile and run the program, see Section 9.12.1 and the *Local Guide*. 
Table 9.9: Closing lines of program wavegd.f.

```fortran
OPEN( UNIT = 1, FILE = 'wavegd_f.dat', STATUS = 'NEW' )
WRITE( UNIT = 1, FMT = 10 )
10 FORMAT( ' Waveguide; Author: David M. Cook; Date: 15 June 1995' )
WRITE( UNIT = 1, FMT = 20 )
20 FORMAT( ' H and E fields in rectangular waveguide' )
WRITE( UNIT = 1, FMT = 30 )
30 FORMAT( ' Program described in CPSUP' )
WRITE( UNIT = 1, FMT = 40 )
40 FORMAT( ' **' )
WRITE( UNIT = 1, FMT = 50 )
50 FORMAT( ' **' )
WRITE( UNIT = 1, FMT = * ) NARR
WRITE( UNIT = 1, FMT = * ) IX
WRITE( UNIT = 1, FMT = * ) IY
WRITE( UNIT = 1, FMT = * ) IZ
DO N = 1, NARR
   DO J = 1, IY
      DO I = 1, IX
         WRITE( UNIT = 1, FMT = * ) H(I,J,1,N)
      ENDDO
   ENDDO
ENDDO
CLOSE( UNIT = 1 )
END
```

```fortran
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 9.11.\(^{53,54,55}\) 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 9.12. A full listing of this program will be found in Section 9.H. The program itself is stored in the file `wavegd.c` in the directory `@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.\(^{56}\) Once the program has been executed, the file `wavegd_c.dat` will exist in the default directory.

---

\(^{53}\)Note that the third index of H is always zero, the only value when the dimension of the array in that index is 1.

\(^{54}\)Note also that, for computational efficiency, various quantities involving only \(x_f\) have been calculated in the outer loop.

\(^{55}\)Note, finally, the use of braces to create blocks of statements that the compiler is to see as single statements.

\(^{56}\)For instructions on how to compile and run the program, see Section 9.12.2 and the Local Guide.
Table 9.10: Opening lines of program wavegd.c.

```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 */
```

Table 9.11: 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);
    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 */
  }
}
```

9.14.3 ... with IDL

The procedure for generating two-dimensional vector arrays is very similar to the method described in Section 9.12.3. However, in this case, we have three two-dimensional arrays, one for each component of the vector fields. The arrays representing the fields in Eq. (9.14) are created using the IDL commands
9.14. CREATING AND STORING TWO-DIMENSIONAL VECTOR ARRAYS

Table 9.12: 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 );
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 );
```

IDL> lugen_grid, x, xrange=[0.0, 2*!pi], nx=29, 
IDL> y, yrange=[0.0, 2*!pi], ny=29
IDL> H = fltarr(30,30,1,3)
IDL> H(*,*,0,0) = sin(x)*cos(y)
IDL> H(*,*,0,1) = -cos(x)*sin(y)
IDL> H(*,*,0,2) = -H(*,*,0,1)

The first two lines produce a grid of values for the evaluation of each component of the fields. Here, x ranges from 0 to 2π, y ranges from 0 to 2π, and the interval is divided into 29 segments. The third line creates a four-dimensional array H consisting of three 30 × 30 × 1 three-dimensional arrays.57

The fourth, fifth, and sixth lines evaluate the two components of H and the one component of E, stored respectively in H(*,*,0,0), H(*,*,0,1), and H(*,*,0,2). Finally, with the statement

IDL> lugen_list, H

we write the data into a file in the desired format.58,59 After the filename—we here choose wavegd_idl.dat—and optional information have been typed in the lugen_list window (see Fig. 9.4)

---

57Remember that dividing a length into 29 segments will require 30 nodes. Remember also that we are treating two-dimensional arrays as three-dimensional arrays in which the third index can assume only one value. Thus, to specify three 30 × 30 two-dimensional arrays we must actually specify three 30 × 30 × 1 three-dimensional arrays.

58The procedure lugen_list invoked here is identical for both two-dimensional scalar and vector arrays and is described in Section 9.12.3.

59A Lawrence-written IDL procedure for reading files written by lugen_list.pro is described in Section 9.16.3.
and ML has been clicked on the ‘SAVE’ button, the data file will be saved 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, 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 $2a$ 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

$$\frac{[E_x(x,y,z), E_y(x,y,z), E_z(x,y,z)]}{q/4\pi\epsilon_0 a^2} = \frac{[x-1,y-1,z]}{[(x-1)^2 + (y-1)^2 + z^2]^{3/2}} - \frac{[x+1,y-1,z]}{[(x+1)^2 + (y-1)^2 + z^2]^{3/2}} + \frac{[x+1,y+1,z]}{[(x+1)^2 + (y+1)^2 + z^2]^{3/2}} - \frac{[x-1,y+1,z]}{[(x-1)^2 + (y+1)^2 + z^2]^{3/2}}$$

(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 $30 \times 30 \times 30 \times 3$ 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.

9.15.1 . . . with FORTRAN

A FORTRAN program to create data for the electric field from the quadrupole described above can be constructed based on the FORTRAN program for three-dimensional scalar arrays. The reader should be familiar with that program, which appears in Section 9.13.1. 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 9.1, is named \texttt{quadpole.f} and contains sections

- Identifying the program.

\footnote{We have supposed indices starting at 1. In languages where indices start at 0, the associations would, of course, be $E(*,*,0,0)$ storing $E_x$, $E(*,*,0,1)$ storing $E_y$, and $E(*,*,0,2)$ storing $E_z$.}
9.15. CREATING AND STORING THREE-DIMENSIONAL VECTOR ARRAYS

As with previous programs, quadpole.f can be copied from the directory $HEAD/fortran 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 quadpole.f.dat will exist in the default directory.

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 9.13, is named 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 [xrange, yrange, zrange] for the interval on each axis and [x0, y0, 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][3], with Ex stored in E[*][*][*][1], Ey stored in E[*][*][*][1], and Ez stored in E[*][*][*][3]. Finally, we introduce variables xfms, xfps, yfms, yfps, and zfs for (x−1)2, (x+1)2, (y−1)2, (y+1)2, and z2, respectively; and variables rmm, rmp, rpm 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 [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 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. For computational efficiency, we introduce variables XFMS, XFPS, YFMS, YFPS, and ZFS for (x−1)2, (x+1)2, (y−1)2, (y+1)2, and z2, respectively; and variables RMM, RMP, RPM and RPP for the four denominators in Eq. 9.15.

- Opening the file, writing the heading lines and the field components into the file, and closing the file.

\footnote{For instructions on how to compile and run the program, see Section 9.12.1 and the Local Guide.}
As with previous programs, *quadpole.c* can be copied from the directory $\text{HEAD/cc}$ 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.\(^{62}\) Once the program has been executed, the file *quadpole.c.dat* will exist in the default directory.

### 9.15.3 ... with IDL

The procedure for using IDL to generate three-dimensional vector arrays is very similar to the method described in Section 9.14.3. In this case, however, we must create three three-dimensional arrays, corresponding to the components of the three-dimensional vector. The array representing the components of the \(E\)-field given by Eq (9.15) is created using the IDL statements

```idl
IDL> lugen_grid, x, xrange=[-2.0, 2.0], nx=29, $  
IDL> y, yrange=[-2.0, 2.0], ny=29, $  
IDL> z, zrange=[-2.0, 2.0], nz=29  
IDL> E = fltarr(30,30,30,3)  
IDL> xp = x + 1.0 & xm = x - 1.0 & yp = y + 1.0 & ym = y - 1.0  
IDL> xps = xp*xp & xms = xm*xm & yps = yp*yp & yms = ym*ym  
IDL> zs = z*z  
IDL> rmm = (xms+yms+zs)^1.5 & rmp = (xms+yps+zs)^1.5  
IDL> rpm = (xps+yms+zs)^1.5 & rpp = (xps+yps+zs)^1.5  
IDL> E(*,*,*,0) = xm*(1.0/rmm - 1.0/rmp) + xp*(1.0/rpp - 1.0/rmp)  
IDL> E(*,*,*,1) = ym*(1.0/rmm - 1.0/rpm) + yp*(1.0/rpp - 1.0/rpm)  
IDL> E(*,*,*,2) = z*(1.0/rmm - 1.0/rmp + 1.0/rpp - 1.0/rpm)
```

The first three lines produce a grid of values for the evaluation of the vector field. Here, \(x\), \(y\), and \(z\) each range from \(-2.0\) to \(2.0\), and the interval is divided into 29 segments. The fourth line creates a four-dimensional array \(E\) consisting of three \(30 \times 30 \times 30\) three-dimensional arrays. The next several lines calculate a number of intermediate quantities to facilitate the final evaluation of the fields. The three lines beginning \(E(*,*,*)\) evaluate the \(E\)-field components. Finally, with the statement

```idl
IDL> lugen_list, E
```

we write the data to a file in the desired format.\(^{63,64}\) After the filename—we here choose *quadpole_idl.dat*—and optional information have been typed in the *lugen_list* window (see Fig. 9.4) and ML has been clicked on the ‘SAVE’ button, the data file will be saved 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.

---

\(^{62}\)For instructions on how to compile and run the program, see Section 9.12.2 and the Local Guide.

\(^{63}\)The procedure *lugen_list* invoked here is identical for both two- and three-dimensional scalar and vector arrays and is described in Section 9.12.3.

\(^{64}\)A Lawrence-written IDL procedure for reading files written by *lugen_list.pro* is described in Section 9.16.3.
9.16. READING FILES

9.16.1 ... with FORTRAN

Data can be read from the files in FORTRAN 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. In this subsection, then, we will develop a FORTRAN program to read the file irradi_f.dat created in Section 9.12.1, understanding that programs to read other files will be similar.

The task involves only a few steps. First, we must open the file for read access with a statement like

```
OPEN( UNIT = 1, FILE = 'irrad_f.dat', STATUS = 'OLD' )
```

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

```
CHARACTER*120 LN1, LN2, LN3, LN4, LN5
READ( UNIT=1, FMT='(A)' ) LN1
READ( UNIT=1, FMT='(A)' ) LN2
READ( UNIT=1, FMT='(A)' ) LN3
READ( UNIT=1, FMT='(A)' ) LN4
READ( UNIT=1, FMT='(A)' ) LN5
```

Next, we read the four dimensions from the file with the statement

```
READ( UNIT=1, FMT='*' ) 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 dimension an array for the data, read the values from the file, and close the file with the statements

```
DIMENSION RI( 50, 50, 1, 1 )
DO N = 1, NARR
   DO K = 1, IZ
      DO J = 1, IY
         DO I = 1, IX
            READ( UNIT=1, FMT='*') RI(I,J,K,N)
         END DO
      END DO
   END DO
END DO
CLOSE( UNIT = 1 )
```

Execution of these statements will recreate the variable RI as it was when the file irradi_f.dat was created in Section 9.12.1. A fully commented and properly ordered listing of the program read_irrad.f is presented in Section 9.1.

---

65Reading 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.
CHAPTER 9. INTRODUCTION TO PROGRAMMING

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. 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;
scanf( 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

```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.M.

---

66Reading 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.
9.16.3 ... with IDL

The Lawrence-generated IDL function {luread_list} parallels the IDL function {lugen_list} first described in Section 9.12.3 and provides the facility for reading into IDL any file created in the format described in Section 9.3.2, whether the files were created with IDL or some other program. We simply execute the statement

{luread_list}, 'FileName', Var

where {FileName} is the name of the file containing the data to be read (including the file type) and Var is the name of the variable into which the data are to be placed when read. For example, the statement

IDL> luread_list, 'irrad_idl.dat', I

will recreate the variable I as it was when the file {irrad_idl.dat} was created in Section 9.12.3. As it executes, {luread_list} displays the descriptive lines that were placed at the beginning of the file when it was created. The procedure {luread_list.pro} that defines this function is presented in Section 9.K.

9.17 References


For IDL, see Section 2.18.

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 9.13 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 9.6 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, 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.

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
Table 9.14: Algorithm for Exercise 9.5.

**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%**
- **ENDIF**
**END_LOOP**

Report **LARGEST%**

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 9.14 is an alternative to the algorithm deduced in (a). Essentially, an alternative method of initializing **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, (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%**, and **C%** satisfying **A%***$^2$ + **B%*$^2$ = **C%*$^2$, subject to the restriction that **A%**, **B%**, and **C%** shall all be smaller than some value **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%**$<=$**A%** or **C%**$<=$**B%**. Express your loops so that these cases (and any others that you can reject a priori) will not even be considered. Optional: For **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 9.15 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 9.16 and choose a suitable name for the procedure.

9.11. Starting with the approximation

\[
\frac{du}{dx}_{x+\frac{1}{2}\Delta x} \approx \frac{u(x+\Delta x) - u(x)}{\Delta x}
\]
Table 9.15: Procedure for Exercise 9.9.

**PROCEDURE**
SCANEND% ← N%
LOOP
  CARD% ← 1%
  Obtain word on card CARD% and store in WORD$
  LATEST.WORD$ ← WORD$
  LATEST.CARD% ← CARD%
  LOOP
    CARD% ← CARD% + 1%
    Obtain word on card CARD% and store in WORD$
    IF WORD$ occurs after LATEST.WORD$
      THEN BEGIN_BLOCK
        LATEST.WORD$ ← WORD$
        LATEST.CARD% ← CARD%
      END_BLOCK
    END_IF
    EXIT_LOOP WHEN CARD% = SCANEND%
END_LOOP
Exchange card LATEST.CARD% with card SCANEND%
SCANEND% ← SCANEND% - 1%
EXIT_LOOP WHEN SCANEND% = 1%
END_LOOP
**END_PROCEDURE**

Deduce the finite difference approximation

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

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.

9.13. (a) Copy *laplace_file.f* from `$HEAD/fortran` to your directory, naming it *laplace15_file.f*. Then compile, link, and run *laplace15_file.f* to generate the file *laplace_f.dat*, which you should rename *laplace15_f.dat*.

(b) Produce the program *laplace29_file.f* by copying *laplace15_file.f* to the new file and editing the new file so that the program, when run, generates a solution on a $29 \times 29$ grid.

(c) Compile, link, and run *laplace29_file.f* and rename the output file to *laplace30_f.dat*.

(d) Import both *laplace15_f.dat* and *laplace30_f.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.f* 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
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

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?
9.14. 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

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

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.

9.15. Consider two circular current loops, each of radius \(a\) and lying with its center on and its plane perpendicular to the \(z\) axis. The trajectory of a particle in three-dimensional space is given parametrically as a function of time \(t\) by the position vector

\[
r = x(t) \mathbf{i} + y(t) \mathbf{j} + z(t) \mathbf{k}
\]

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 one tool but will visualize the trajectory with another tool. You must generate the data using tool that does not have much computational capability. Thus, you must generate the data using

- 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

\[
r = \cos t \mathbf{i} + \sin t \mathbf{j} + 0.1t \mathbf{k}
\]
which describes the path followed by a charged particle in a constant magnetic field along the \( z \) axis.

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

\[
p(x, y, z, t) = A \sin \frac{\ell \pi x}{a} \sin \frac{m \pi y}{a} \sin \frac{n \pi z}{a} \cos \omega t, 
\]

where \( \ell, m, \) 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 \( \ell, m, \) and \( n \).

9.19. A point charge of strength \( q \) is located on the \( y \) axis at \( r_+ = a \hat{j} \) and a point charge of strength \(-q\) is located on the \( x \) axis at \( r_- = -a \hat{i} \). 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

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

Expressing coordinates in terms of the dimensionless variables \( \tilde{x} = x/a \) and \( \tilde{y} = y/a \) 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 \( x \) 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 \( \mathbf{\omega} \) is given in terms of the angular velocity and the position vector \( \mathbf{r} \) by the expression

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

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

\[
\mathbf{E}(x, y, z) = \frac{q}{4\pi \epsilon_0} \left[ \frac{x \hat{i} + y \hat{j} + (z - a) \mathbf{\hat{k}}}{[x^2 + y^2 + (z - a)^2]^{3/2}} - \frac{x \hat{i} + y \hat{j} + (z + a) \mathbf{\hat{k}}}{[x^2 + y^2 + (z + a)^2]^{3/2}} \right]
\]
Expressing coordinates in terms of the dimensionless variables $\bar{x} = x/a$, $\bar{y} = y/a$, and $\bar{z} = z/a$ and measuring $E$ 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°40′ N, 73°45′ W]; Grand Junction, CO, is at [39°5′ N, 108°33′ W]; Los Angeles, CA, is at [34°3′ N, 118°15′ W]; Appleton, WI, is at [44°16′ N, 88°25′ W]; Calcutta India, is at [22°32′ N, 88°20′ E]; Sydney, Australia, is at [33°52′ S, 151°12′ E]; Paris, France, is at [48°49′ N, 2°29′ E]; and Stockholm, Sweden, is at [59°21′ N, 18°4′ E].
9.A  Listing of irrad.f

PROGRAM IRRAD

PARAMETER( NARR = 1, IX = 50, IY = 50, IZ = 1 )

!***** DIMENSION VARIABLES, EVALUATE CONSTANTS *****

DIMENSION RI( IX, IY ) ! For irradiance
THREEPI = 3.0*3.1415926535 ! Set constant
DX = 2.0*THREEPI/(IX-1.0) ! Set increments
DY = DX

!***** EVALUATE IRRADIANCE *****

DO J = 1, IY
  YF = (J-1)*DY - THREEPI
  DO I = 1, IX
    XF = (I-1)*DX - THREEPI
    RI(I,J) = ( (SIN(XF)/XF)**2 ) * ( (SIN(YF)/YF)**2 )
  END DO
END DO

!***** OPEN FILE FOR WRITING, WRITE HEADING LINES TO FILE *****

OPEN( UNIT = 1, FILE = 'irrad_f.dat', STATUS = 'NEW' )
WRITE( UNIT = 1, FMT = 10 )
10 FORMAT( ' Irradiance; Author: David M. Cook; Date: 15 June 1995' )
WRITE( UNIT = 1, FMT = 20 )
20 FORMAT( ' Fraunhofer diffraction at square aperture' )
WRITE( UNIT = 1, FMT = 30 )
30 FORMAT( ' Program described in CPSUP' )
WRITE( UNIT = 1, FMT = 40 )
40 FORMAT( ' **' )
WRITE( UNIT = 1, FMT = 50 )
50 FORMAT( ' **' )
WRITE( UNIT = 1, FMT = * ) NARR
WRITE( UNIT = 1, FMT = * ) IX
WRITE( UNIT = 1, FMT = * ) IY
WRITE( UNIT = 1, FMT = * ) IZ

!***** WRITE IRRADIANCE TO FILE, CLOSE FILE *****

DO J = 1, IY
  DO I = 1, IX
    WRITE( UNIT = 1, FMT = * ) RI(I, J)
  END DO
END DO
CLOSE( UNIT = 1 )

END
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()
{


***** DECLARE VARIABLES, EVALUATE CONSTANTS *****/

    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 *****/

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

fprintf( fptr, "%d\n", ydim );
fprintf( 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 );
}
9.C Listing of lugen_list.pro (IDL)

pro lugen_list, arr

; Lugen_list and lugen_list_event work together to facilitate writing data stored in an array in IDL into an ASCII file in a standard format for purposes both of saving the data and of transferring it to other programs for further examination. The array to be written to the file is communicated to lugen_list through its one argument arr.

; Establish a COMMON block named tblock for storing the array and the ID's of several text-entry boxes so they can be communicated to the event handler lugen_list_event.

COMMON tblock, array, fname, ti, au, da, c1, c2, c3, c4

; Move array supplied in the input variable as arr to the COMMON block.
array=arr

; Set default filename
filename='untitled.dat'

; Create outermost widget named base (i.e., with its unique ID stored in base).
base=widget_base(title=" L U G E N _ L I S T ",
                 /row, xoffset=150, yoffset=250)

; Create child widgets lcol and rcol within base.
lcol=widget_base(base, /column, /frame)
rcol=widget_base(base, /column, /frame, space=50)

; Create widget with ID stored in b within lcol and, within b, create labeled text-entry boxes. Note that the text-entry boxes have ID's stored in fname and ti, whose values reside in the COMMON block. Note also that the uvalues 'FILENAME' and 'TITLE' are user-assigned here for later convenient identification of the referenced widgets.

b=widget_base(lcol, /row, /frame)
fnl=widget_label(b, value='Filename:')
fname=widget_text(b, /editable, xsize=25, ysize=1, value=filename,$ uvalue='FILENAME')
til=widget_label(b, value='Title:')
ti=widget_text(b, /editable, xsize=25, ysize=1, uvalue='TITLE')

; Repeat, creating labeled text entry windows for the author and the date. Since we have no further use for the ID previously written
; into b, we can here reuse the variable b.

b=widget_base(lcol, /row, /frame)
au=widget_label(b, value='Author : ')
au=widget_text(b, /editable, xsize=25, ysize=1, uvalue='AUTHOR')
dal=widget_label(b, value='Date : ')
da=widget_text(b, /editable, xsize=25, ysize=1, uvalue='DATE')

; Repeat, creating labeled text entry windows for four lines of comments.

b=widget_base(lcol, /row, /frame)
c1l=widget_label(b, value='Comments 1: ')
c1=widget_text(b, /editable, xsize=65, ysize=1, value='**', uvalue='COMM1')
b=widget_base(lcol, /row, /frame)
c2l=widget_label(b, value='Comments 2: ')
c2=widget_text(b, /editable, xsize=65, ysize=1, value='**', uvalue='COMM2')
b=widget_base(lcol, /row, /frame)
c3l=widget_label(b, value='Comments 3: ')
c3=widget_text(b, /editable, xsize=65, ysize=1, value='**', uvalue='COMM3')
b=widget_base(lcol, /row, /frame)
c4l=widget_label(b, value='Comments 4: ')
c4=widget_text(b, /editable, xsize=65, ysize=1, value='**', uvalue='COMM4')

; Create SAVE and CANCEL buttons within rcol.

button1=widget_button(rcol, value='Save', uvalue='SAVE')
button1=widget_button(rcol, value='Close', uvalue='CLOSE')

; Display the created widget.

widget_control, base, /realize

; Invoke xmanager to sense the mouse events occurring within the base widget. With each event, xmanager calls lugen_list_event to provide the proper response.

xmanager, 'lugen_list', base

END
9.D  Listing of lugen_list_event.pro (IDL)

```idl
pro lugen_list_event, event

; This pro-file works in conjunction with the profile lugen_list.
; Everytime an event (mouse click) occurs in the base widget created by
; lugen_list, the IDL routine xmanager, which is invoked at the end of
; lugen_list, itself invokes lugen_list_event and supplies the input
; value for the variable event. Event is a structured variable
; containing information about the event that has occurred. That
; information is used here to determine what action should be taken
; in response to the event.

; Create COMMON block parallel to the one in lugen_list.
COMMON tblock, array, fname, ti, au, da, c1, c2, c3, c4

; Fetch the uvalue assigned in lugen_list to identify the particular
; widget in which the event occurred, storing that uvalue in the
; variable eventval, and then execute the proper response determined
; by the uvalue associated with the event that occurred.

widget_control, event.id, get_uvalue=eventval

case eventval of

    'FILENAME':
    'TITLE':
    'AUTHOR':
    'DATE':
    'COMM1':
    'COMM2':
    'COMM3':
    'COMM4':

    ; Respond to mouse click on SAVE button by reading data in
    ; base widget, processing it, writing it to the specified
    ; output file, and terminating execution of lugen_list. Note that
    ; the widget ID’s written into the COMMON block by lugen_list are
    ; here read out of that COMMON block to identify the widget
    ; to be read by each widget_control statement.

    'SAVE' : begin

        ; Read data in text-entry windows.

        widget_control, fname, get_value=filename
        widget_control, ti, get_value=title
        widget_control, au, get_value=author
        widget_control, da, get_value=date
```
; Determine dimensions of array

dim = size(array)
narr = 1
xdim = fix(dim[1])
ydim = fix(dim[2])
zdim = fix(dim[3])
case dim[0] of
  2: zdim = 1
  3:
  4: narr = fix(dim[4])
endcase

; Write output file, inform user of success,
; return to IDL main level

openw, 1, filename(0)
printf, 1, title + ' Author: ' + author + ' Date: ' + date
printf, 1, comment1
printf, 1, comment2
printf, 1, comment3
printf, 1, comment4
printf, 1, narr
printf, 1, xdim
printf, 1, ydim
printf, 1, zdim
for n = 0, narr-1 do$
  for k = 0, zdim-1 do$
    for j = 0, ydim-1 do$
      for i = 0, xdim-1 do$
        printf, 1, array[i,j,k,n]

close, 1

print, "File ", filename(0), ", saved successfully!"

widget_control, event.top, /destroy
end

; Respond to mouse click on CLOSE button.

'CLOSE' : widget_control, event.top, /destroy

endcase

END
9.E Listing of pdens.f

PROGRAM PDENS

PARAMETER( NARR=1, IX=30, IY=30, IZ=30 )
PARAMETER( XRANGE=20.0, YRANGE=20.0, ZRANGE=20.0 )
PARAMETER( X0=-10.0, Y0=-10.0, Z0=-10.0 )

!****** DIMENSION VARIABLES; EVALUATE CONSTANTS *****
DIMENSION P( IX, IY, IZ ) ! For probability density
TWOPI = 2.0*3.1415926536 ! Set other constants
PFACOR = 1/(TWOPI*(27.0*27.0*27.0))
DX = XRANGE/(IX-1.0) ! Set increments
DY = YRANGE/(IY-1.0)
DZ = ZRANGE/(IZ-1.0)

!****** EVALUATE PROBABILITY DENSITY *****

DO K = 1, IZ
    ZF = (K-1)*DZ + Z0
    ZFS = ZF*ZF
    DO J = 1, IY
        YF = (J-1)*DY + Y0
        YFS = YF*YF
        DO I = 1, IX
            XF = (I-1)*DX + X0
            XFS = XF*XF
            RHOS = XF*XF + YF*YF + ZF*ZF
            RHO = SQRT(RHOS)
            P(I,J,K) = PFACTOR * EXP( -2.0*RHO/3.0 ) * 
                       ( 9.0*ZFS*ZFS - 6.0*ZFS*RHOS + RHOS*RHOS )
        END DO
    END DO
END DO

!****** OPEN FILE FOR WRITING *****
OPEN( UNIT = 1, FILE = 'pdens_f.dat', STATUS = 'NEW' )

!****** WRITE HEADING LINES TO FILE *****
WRITE( UNIT = 1, FMT = 10 )
10 FORMAT( ' Prob. Dens.; Author: Michael Stenner; Date: 6/21/95' )
WRITE( UNIT = 1, FMT = 20 )
20 FORMAT( ' The probability density for the electron in the' )
WRITE( UNIT = 1, FMT = 30 )
30 FORMAT( ' (n,l,m) = (3,2,0) state in the hydrogen atom.' )
WRITE( UNIT = 1, FMT = 40 )
40 FORMAT( ' Program described in CPSUP' )
WRITE( UNIT = 1, FMT = 50 )
50 FORMAT( ' **' )
WRITE( UNIT = 1, FMT = * ) NARR
WRITE( UNIT = 1, FMT = * ) IX
WRITE( UNIT = 1, FMT = * ) IY
WRITE( UNIT = 1, FMT = * ) IZ

!***** WRITE PROBABILITY DENSITY TO FILE *****

DO K = 1, IZ
  DO J = 1, IY
    DO I = 1, IX
      WRITE( UNIT = 1, FMT = * ) P(I,J,K)
    END DO
  END DO
END DO

!***** CLOSE FILE *****

CLOSE( UNIT = 1 )
END
CHAPTER 9. INTRODUCTION TO PROGRAMMING

9.F Listing of pdens.c

/* 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()
{

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

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;
        } /* End of for( x=0; x<xdim; x++ ) */
    } /* End of for( y=0; y<ydim; y++ ) */
} /* End of for( z=0; z<zdim; z++ ) */

*/
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 );
9.G Listing of wavegd.f

PROGRAM WAVEGD

PARAMETER( NARR = 3, IX = 30, IY = 30, IZ = 1 )

!****** DIMENSION VARIABLES; EVALUATE CONSTANTS *****!

DIMENSION H( IX, IY, IZ, NARR ) ! For field components

TWOPI = 2.0*3.1415926535 ! Set constant
DX = TWOPI/(IX-1.0) ! Set increments
DY = DX

!****** EVALUATE FIELD COMPONENTS *****

DO I = 1, IX
  XF = (I-1)*DX
  SXF = SIN(XF)
  CXF = COS(XF)
  DO J = 1, IY
    YF = (J-1)*DY
    SYF = SIN(YF)
    CYF = COS(YF)
    H(I,J,1,1) = SXF * CYF ! Hx
    H(I,J,1,2) = -CXF * SYF ! Hy
    H(I,J,1,3) = -H(I,J,1,2) ! Ez
  END DO
END DO

!****** OPEN FILE FOR WRITING *****

OPEN( UNIT = 1, FILE = 'wavegd_f.dat', STATUS = 'NEW' )

!****** WRITE HEADING LINES TO FILE *****

WRITE( UNIT = 1, FMT = 10 )
10 FORMAT( ' Waveguide; Author: David M. Cook; Date: 15 June 1995' )
WRITE( UNIT = 1, FMT = 20 )
20 FORMAT( ' H and E fields in rectangular waveguide' )
WRITE( UNIT = 1, FMT = 30 )
30 FORMAT( ' Program described in CPSUP' )
WRITE( UNIT = 1, FMT = 40 )
40 FORMAT( ' **' )
WRITE( UNIT = 1, FMT = 50 )
50 FORMAT( ' **' )
WRITE( UNIT = 1, FMT = * ) NARR
WRITE( UNIT = 1, FMT = * ) IX
WRITE( UNIT = 1, FMT = * ) IY
WRITE( UNIT = 1, FMT = * ) IZ

!****** WRITE FIELD COMPONENTS TO FILE *****


DO N = 1, NARR  ! Write H to file
   DO J = 1, IY
      DO I = 1, IX
         WRITE( UNIT = 1, FMT = * ) H(I,J,1,N)
      END DO
   END DO
END DO

!***** CLOSE FILE *****

CLOSE( UNIT = 1 )

END
9.H Listing of wavegd.c

/* 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 *****/
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 );

/
***** WRITE FIELD TO FILE *****/

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 *****/

fclose( fptr );
CHAPTER 9. INTRODUCTION TO PROGRAMMING

9.I Listing of quadpole.f

PROGRAM QUADPOLE

PARAMETER( NARR=3, IX=30, IY=30, IZ=30 )
PARAMETER( XRANGE=4.0, YRANGE=4.0, ZRANGE=4.0 )
PARAMETER( X0=-2.0, Y0=-2.0, Z0=-2.0)

!***** DIMENSION VARIABLES; EVALUATE CONSTANTS *****

DIMENSION E( IX, IY, IZ, NARR ) ! For electric field
DX = XRANGE/(IX-1.0) ! Set increments
DY = YRANGE/(IY-1.0)
DZ = ZRANGE/(IZ-1.0)

!***** EVALUATE ELECTRIC FIELD *****

DO K = 1, IZ
  ZF = (K-1)*DZ + Z0
  ZFS = ZF*ZF
  DO J = 1, IY
    YF = (J-1)*DY + Y0
    YFM = YF - 1.0
    YFP = YF + 1.0
    YFMS = YFM*YFM
    YFPS = YFP*YFP
    DO I = 1, IX
      XF = (I-1)*DX + X0
      XFM = XF - 1.0
      XFP = XF + 1.0
      XFMS = XFM*XFMS
      XFPS = XFP*XFPS
      RMM = (XFMS + YFMS + ZFS)**1.5
      RMP = (XFMS + YFPS + ZFS)**1.5
      RPP = (XFPS + YFPS + ZFS)**1.5
      RPM = (XFPS + YFMS + ZFS)**1.5
      E(I,J,K,1) = XFM * (1.0/RMM - 1.0/RMP) ! Ex
      + XFP * (1.0/RPP - 1.0/RPM) ! Ey
      + YFM * (1.0/RMM - 1.0/RPM) ! Ez
      + YFP * (1.0/RPP - 1.0/RPM)
    END DO
  END DO
END DO

! OPEN FILE FOR WRITING *****

OPEN( UNIT = 1, FILE = 'quadpole_f.dat', STATUS = 'NEW' )

! WRITE HEADING LINES TO FILE *****

WRITE( UNIT = 1, FMT = 10 )
10 FORMAT( ' Quadpole.; Author: Michael Stenner; Date: 6/26/95' )
WRITE( UNIT = 1, FMT = 20 )
20 FORMAT( ' The electric field of a quadrupole with charges of +1' )
WRITE( UNIT = 1, FMT = 30 )
30 FORMAT( ' at (1,1) and (-1,-1) and charges of -1 at (1,-1) and' )
WRITE( UNIT = 1, FMT = 40 )
40 FORMAT( ' (-1,1). Values are dimensionless vectors. N=1 denotes')
WRITE( UNIT = 1, FMT = 50 )
50 FORMAT( ' x components; N=2, y components; and N=3, z components')
WRITE( UNIT = 1, FMT = * ) NARR
WRITE( UNIT = 1, FMT = * ) IX
WRITE( UNIT = 1, FMT = * ) IY
WRITE( UNIT = 1, FMT = * ) IZ

****** WRITE ELECTRIC FIELD TO FILE *****

DO N = 1, NARR
   DO K = 1, IZ
      DO J = 1, IY
         DO I = 1, IX
            WRITE( UNIT = 1, FMT = * ) E(I,J,K,N)
         END DO
      END DO
   END DO
END DO

****** CLOSE FILE *****

CLOSE( UNIT = 1 )
END
9.J  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 xo -2.0 /* Set starting value of x */
#define yo -2.0 /* Set starting value of y */
#define zo -2.0 /* Set starting value of z */

main()
{
    /***** DECLARE VARIABLES; EVALUATE CONSTANTS *****/
    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, rpm, 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 + zo;
        zfs = zf*zf;
        for( y=0; y<ydim; y++ )
        {
            yf = y * dy + yo;
            yfm = yf - 1.0;
            yfp = yf + 1.0;
            yfms = yfm*yfm;
            yfps = yfp*yfp;
            for( x=0; x<xdim; x++ )
            {
                xf = x * dx + xo;
                xfm = xf - 1.0;
                xfp = xf + 1.0;
                xfms = xfm*xfm;
            } /* End for x */
        } /* End for y */
    } /* End for z */
xfps = xfp*xfp;
rm = pow(xfms + yfms + zfs, 1.5);
rmp = pow(xfms + yfps + zfs, 1.5);
rpp = pow(xfps + yfps + zfs, 1.5);
rpm = 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/rmp);
E[x][y][z][2] = zf*(1.0/rmm - 1.0/rmp) /* Ez */
            + 1.0/rpp - 1.0/rpm);
}
}
}
}

***** OPEN FILE IN WRITE MODE; GET POINTER *****
fptr = fopen("quadpole_c.dat", "w");

***** WRITE HEADING LINES TO FILE *****
fprintf(fp,"Quadrapole; Author: Chris Schmidt;");
fprintf(fp," Date: 27 June 1995
");
fprintf(fp,"The electric field from 4 point charges positioned
");
fprintf(fp,"on the corners of a square.
");
fprintf(fp,"Program described in CPSUP
");
fprintf(fp,"**
");

fprintf(fp,"%d
", narr);
fprintf(fp,"%d
", zdim);
fprintf(fp,"%d
", ydim);
fprintf(fp,"%d
", xdim);

***** WRITE ELECTRIC FIELD TO FILE *****
for( n=0; n<narr; n++ )
{
    for( z=0; z<zdim; z++ )
    {
        for( y=0; y<ydim; y++)
        {
            for( x=0; x<xdim; x++ )
            {
                fprintf(fp,"%e
", E[x][y][z][n]);
            }
        }
    }
}

/* CLOSE FILE *****
fclose(fp);
}
9.K Listing of luread_list.pro (IDL)

```idl
pro luread_list, filename, U

; This pro-file reads a file created in the format described in
; CPSUP and stores the array internal to IDL. It is invoked
; with the command
;
;   luread_list, 'filename', U
;
; where 'filename' specifies the file and U identifies the name of the
; variable into which the data will be read.
;

;***** Block 1: Print starting message
; Specify string variables for reading first lines of file *****

print, format = '($,"Reading file")'
head1 = ''; head2 = ''; head3 = ''; head4 = ''; head5 = ''

;***** Block 2: Open file for reading, read headers and dimensions
; into IDL, and write headers to terminal *****

openr, 1, filename
readf, 1, head1 & print, format = '($,".")'
readf, 1, head2 & print, format = '($,".")'
readf, 1, head3 & print, format = '($,".")'
readf, 1, head4 & print, format = '($,".")'
readf, 1, head5 & print, format = '($,".")'
readf, 1, narr & narr = fix(narr) & print, format = '($,".")'
readf, 1, xdim & xdim = fix(xdim) & print, format = '($,".")'
readf, 1, ydim & ydim = fix(ydim) & print, format = '($,".")'
readf, 1, zdim & zdim = fix(zdim) & print, format = '($,".")'
print, " ."

print, " ", head1 & print, " ", head2 & print, " ", head3
print, " ", head4 & print, " ", head5

; ***** Block 3: Read U into IDL and close file *****

U = fltarr( xdim, ydim, zdim, narr )
for n = 0, narr-1 do $
    for k = 0, zdim-1 do $
        for j = 0, ydim-1 do $
            for i = 0, xdim-1 do $
                begin readf,1, tmp & U[i,j,k,n] = tmp & endfor

close, 1

;***** Block 4: Print reassuring message *****

print, ".........File ", filename, ": read successfully!"
```

end
9.L Listing of read_irrad.f

PROGRAM READ_IRRAD

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

C ***** Set variables to store comments, array *****

CHARACTER*120 LN1, LN2, LN3, LN4, LN5
DIMENSION RI( 50, 50, 1, 1 )

C ***** Open file for reading; read comments and dimensions *****

OPEN( UNIT = 1, FILE = 'irrad_f.dat', STATUS = 'OLD' )
READ( UNIT=1, FMT='(A)' ) LN1
READ( UNIT=1, FMT='(A)' ) LN2
READ( UNIT=1, FMT='(A)' ) LN3
READ( UNIT=1, FMT='(A)' ) LN4
READ( UNIT=1, FMT='(A)' ) LN5
READ( UNIT=1, FMT='*' ) NARR, IX, IY, IZ

C ***** Read array in proper order; close file *****

DO N = 1, NARR
  DO K = 1, IZ
    DO J = 1, IY
      DO I = 1, IX
        READ( UNIT=1, FMT='*' ) RI(I,J,K,N)
      END DO
    END DO
  END DO
END DO
CLOSE( UNIT = 1 )

C ***** Coding that processes the array as desired would then follow
C at this point in the program.

END
9.M  Listing of read_irrad.c

/* 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 dimensions of array */
  int n, i, j, k; /* For loop indices */
  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<iy; 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

The Numerical Recipes package\textsuperscript{1,2} contains numerous subroutines implementing a wide variety of numerical algorithms and provides a valuable and 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: \emph{Numerical Recipes: The Art of Scientific Computing} (hereafter referred to as the main source book) and \emph{Numerical Recipes: Example Book} (the example source book).\textsuperscript{3} 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.\textsuperscript{4}

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 $\$NRHEAD$.\textsuperscript{5} As illustrated in Fig. 10.1, the full tree contains three subdirectories named \texttt{recipes.f}, \texttt{recipes.c-ansi}, and \texttt{recipes.c-kr}, though not all of these subdirectories will be present at every site. Each of these subdirectories in turn contains several (sub)subdirectories, the most significant of which are named \texttt{recipes}, \texttt{demo}, and (for the C directories) \texttt{include}. Further, the directory \texttt{demo} contains several still lower level directories, including \texttt{data} and \texttt{src}. For each language,

- The directory $\$NRHEAD$/\texttt{recipes\_language}/\texttt{recipes}$ contains all of the recipes themselves.
  
  These recipes are described in detail in the main source books.

\textsuperscript{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 \textit{Local Guide}.

\textsuperscript{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.

\textsuperscript{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.

\textsuperscript{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++.

\textsuperscript{5}See the \textit{Local Guide} for the translation of this symbol at your site.
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.

10.2 Using FORTRAN Recipes

In FORTRAN, the files in the FORTRAN-77 subdirectory recipes_f/recipes have names with filetype .f (or .for for some operating systems) and those in the FORTRAN-90 subdirectory recipes_f-90/recipes have names with filetype .f90. 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.f (or, possibly, flmoon.for). The associated demonstration file is named xflmoon.f (or, possibly, xflmoon.for). To use the demonstration program,
1. We copy the program xflmoon.f into our default directory with the statement:\(^6\)
\[\text{cp } \$NRHEAD/recipes_f/demo/src/xflmoon.f .\]

2. We copy the associated recipe flmoon.f into our default directory with the statement
\[\text{cp } \$NRHEAD/recipes_f/recipes/flmoon.f .\]

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:\(^7\)
\[f77 -o xflmoon.xf xflmoon.f flmoon.f\]

If this statement generates no error messages, then no further recipes are needed.\(^8\) More likely, others will be needed. They will be identified in the error messages as “unresolved text symbols”. In the present case, the symbols julday and caldat will be identified, so we need the statements
\[\quad \text{cp } \$NRHEAD/recipes_f/recipes/calldat.f .\]
\[\quad \text{cp } \$NRHEAD/recipes_f/recipes/julday.f .\]

to copy these two additional files into our default directory.

A more gentle way to identify needed additional files entails examining the listings of the program xflmoon.f and the recipe flmoon.f, noting the statements that invoke subroutines flmoon and caldat and function julday. Strictly, we should then examine the listings of any subroutines so identified to see if they in turn invoke still other subroutines.

4. We compile and load the program with the statement
\[f77 -o xflmoon.xf xflmoon.f flmoon.f caldat.f julday.f\]

The order of the filenames in this statement is irrelevant. Note, incidentally, that the command \(f77\) 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:\(^9\)
\[./xflmoon.xf\]

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 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 11.14.2, 13.13.2, and 14.14.2.

\(^6\)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.

\(^7\)Throughout this book, we shall assume that we are using the FORTRAN-77 routines. Presumably, the FORTRAN-90 and FORTRAN-95 routines are used in much the same way, though the compile command in UNIX will be \(f90\) and \(f95\), respectively.

\(^8\)The routine xflmoon.f invokes the FORTRAN pause statement, which apparently has been discontinued in more recent FORTRAN compilers. A warning will be generated, though it will not prevent compilation of the program if all other requirements are met.

\(^9\)Again, 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.
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\(^{10}\)
   ```
   cp $NRHEAD/recipes_c-ansi/demo/src/xflmoon.c .
   ```

2. We copy the associated recipe *flmoon.c* into our default directory with the statement
   ```
   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
   ```
   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
   ```
   cp $NRHEAD/recipes_c-ansi/include/nr.h .
   ```
   Then, a repeat compilation with the statement
   ```
   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
   ```
   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
   ```
   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`.

\(^{10}\)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*.

\(^{11}\)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.
A more gentle way to identify needed additional files entails examining the listings of the program \texttt{xflmoon.c} and the recipe \texttt{flmoon.c}, noting all non-standard include files and all non-standard procedure calls. In the present case, examining \texttt{xflmoon.c} reveals the inclusion of \texttt{nr.h} and calls to \texttt{julday} and \texttt{caldat} while examination of \texttt{flmoon.c} reveals invocation of \texttt{nrerror} (which is defined in \texttt{nrutil.c}), \texttt{sin}, \texttt{floor}, and \texttt{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\textsuperscript{12}

\begin{verbatim}
cc -o xflmoon.xc xflmoon.c flmoon.c julday.c caldat.c nrutil.c -lm
\end{verbatim}

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

5. We run the program with the statement\textsuperscript{13}

\begin{verbatim}
./xflmoon.xc
\end{verbatim}

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

10.1. The recipe \texttt{julday.f} 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 \texttt{xjulday.f}, the primary recipe \texttt{julday.f}, and any necessary additional files from the \textit{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. \textit{Note}: The file \texttt{DATES.DAT} needed by the program can be copied from the directory $\texttt{NRHEAD/recipes.f/demo/results}$.

10.2. The recipe \texttt{avevar.f} 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 \texttt{xavevar.f}, the primary recipe \texttt{avevar.f}, and any necessary additional files from the \textit{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.

\textsuperscript{12}This statement may generate a warning about conflicting data types for one or more variables. Warnings, however, do not prevent successful compilation.

\textsuperscript{13}Again, 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.
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.
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\(^1\) \(-mg \mathbf{k}\) and the viscous force \(F_v\), from the medium in which it moves. The former is directed downward and the latter is directed

\(^1\)We choose a coordinate system in which positive \(z\) is directed upward, and we take \(m\) and \(g\) to be positive.
opposite to the velocity $v$. Usually, the magnitude of the viscous force is a function of the speed with which the projectile moves, symbolically $|F_v| = f(|v|)$. In general, if the projectile has mass $m$, Newton’s second law yields the equation of motion

$$m \frac{d^2 \mathbf{r}}{dt^2} = -mg \, \hat{k} + F_v(|v|) = -mg \, \hat{k} - f(|v|) \frac{v}{|v|}$$  \hspace{1cm} (11.1)$$

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

$$\mathbf{r}(0) = \mathbf{r}_0 = x_0 \, \hat{i} + y_0 \, \hat{j} + z_0 \, \hat{k} \quad ; \quad \mathbf{v}(0) = \mathbf{v}_0 = v_{x_0} \, \hat{i} + v_{y_0} \, \hat{j} + v_{z_0} \, \hat{k}$$  \hspace{1cm} (11.2)$$

Of course, we also need to know the precise dependence of the function $f(|v|)$ 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, $f(|v|) = b \, |v|$ ($b$ a positive constant), and the equation of motion reduces to

$$m \frac{d^2 \mathbf{r}}{dt^2} = -mg \, \hat{k} - b \, v = -mg \, \hat{k} - b \, \frac{d\mathbf{r}}{dt}$$  \hspace{1cm} (11.3)$$
or, in component form, to

$$m \frac{d^2 x}{dt^2} = -b \, \frac{dx}{dt} \quad ; \quad m \frac{d^2 y}{dt^2} = -b \, \frac{dy}{dt} \quad ; \quad m \frac{d^2 z}{dt^2} = -mg - b \, \frac{dz}{dt}$$  \hspace{1cm} (11.4)$$

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, $f = b \, |v|^2$ ($b$ a positive constant, though not the same constant as in the previous paragraph). This time, the equation of motion reduces to

$$m \frac{d^2 \mathbf{r}}{dt^2} = -mg \, \hat{k} - b \, |v| \, v = -mg \, \hat{k} - b \, \frac{d\mathbf{r}}{dt} \sqrt{\left(\frac{dx}{dt}\right)^2 + \left(\frac{dy}{dt}\right)^2 + \left(\frac{dz}{dt}\right)^2}$$  \hspace{1cm} (11.5)$$
or, in component form, to

$$m \begin{bmatrix} \frac{d^2 x}{dt^2} & \frac{d^2 y}{dt^2} & \frac{d^2 z}{dt^2} \end{bmatrix} = [0, 0, -mg] - b \begin{bmatrix} \frac{dx}{dt} & \frac{dy}{dt} & \frac{dz}{dt} \end{bmatrix} \sqrt{\left(\frac{dx}{dt}\right)^2 + \left(\frac{dy}{dt}\right)^2 + \left(\frac{dz}{dt}\right)^2}$$  \hspace{1cm} (11.6)$$

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

$$m \frac{d^2 z}{dt^2} = -mg - b \, \frac{dz}{dt} \left|\frac{dz}{dt}\right|$$  \hspace{1cm} (11.7)$$
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 \( \tau = x/a \), \( \eta = y/a \), and \( \zeta = z/a \) (i.e., \( \overline{r} = r/a \)), we find that Eqs. (11.3) and (11.5) become
\[
\frac{d^2 \overline{r}}{dt^2} = -\frac{g}{a} \hat{k} - \frac{b}{m} \frac{d\overline{r}}{dt}
\] (11.8)
and
\[
\frac{d^2 \overline{\eta}}{dt^2} = -\frac{g}{a} \hat{k} - \frac{ba}{m} \hat{\alpha} \frac{d\overline{\eta}}{dt} \left[ \left( \frac{d\overline{\tau}}{dt} \right)^2 + \left( \frac{d\overline{\eta}}{dt} \right)^2 + \left( \frac{d\overline{\zeta}}{dt} \right)^2 \right]
\] (11.9)
respectively. Next, we recognize that \( \sqrt{g/a} \) is dimensionally a frequency. Hence, the variable \( \overline{t} = t \sqrt{g/a} \) provides a suitable rescaling of the independent variable. In terms of \( \overline{t} \), Eqs. (11.8) and (11.9) become
\[
\frac{d^2 \overline{r}}{d\overline{t}^2} = -\hat{k} - \frac{b}{m} \sqrt{\frac{a}{g}} \frac{d\overline{r}}{d\overline{t}}
\] (11.10)
and
\[
\frac{d^2 \overline{\eta}}{d\overline{t}^2} = -\hat{k} - \overline{\beta} \frac{d\overline{\eta}}{d\overline{t}} \sqrt{\frac{a}{g}} \left[ \left( \frac{d\overline{\tau}}{d\overline{t}} \right)^2 + \left( \frac{d\overline{\eta}}{d\overline{t}} \right)^2 + \left( \frac{d\overline{\zeta}}{d\overline{t}} \right)^2 \right]
\] (11.11)
Finally, we introduce the symbol \( \overline{\beta} \) 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. In the end, the equations in dimensionless form are
\[
\frac{d^2 \overline{r}}{d\overline{t}^2} = -\hat{k} - \overline{\beta} \frac{d\overline{r}}{d\overline{t}}
\] (11.12)
and
\[
\frac{d^2 \overline{\eta}}{d\overline{t}^2} = -\hat{k} - \overline{\beta} \frac{d\overline{\eta}}{d\overline{t}} \sqrt{\frac{a}{g}} \left[ \left( \frac{d\overline{\tau}}{d\overline{t}} \right)^2 + \left( \frac{d\overline{\eta}}{d\overline{t}} \right)^2 + \left( \frac{d\overline{\zeta}}{d\overline{t}} \right)^2 \right]
\] (11.13)
These equations are, of course, to be solved subject to the dimensionless initial conditions
\[
\overline{r}(0) = \frac{r_0}{a} ; \quad \overline{v}(0) = \frac{d\overline{r}}{d\overline{t}} (0) = \frac{d(r/a)}{d(t \sqrt{g/a})}(0) = \frac{v_0}{\sqrt{g/a}}
\] (11.14)
from which we conclude that dimensionless velocities are measured in units of \( \sqrt{g/a} \).

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
\[
m \frac{dv_x}{dt} = -bv_x ; \quad m \frac{dv_y}{dt} = -bv_y ; \quad m \frac{dv_z}{dt} = -mg - bv_z
\] (11.15)
the three members of Eq. (11.6) become
\[
m \left[ \frac{dv_x}{dt}, \frac{dv_y}{dt}, \frac{dv_z}{dt} \right] = [0, 0, -mg] - b [v_x, v_y, v_z] \sqrt{v_x^2 + v_y^2 + v_z^2}
\] (11.16)
and Eq. (11.7) becomes
\[
m \frac{dv_z}{dt} = -mg - bv_z |v_z|
\] (11.17)
Once these first-order equations have been solved, the components of the position vector can then be found by solving the differential equations
\[
\frac{dx}{dt} = v_x ; \quad \frac{dy}{dt} = v_y ; \quad \frac{dz}{dt} = v_z
\] (11.18)
\footnote{Using \([\ldots]\) to indicate "the dimensions of \(\ldots\), we argue \([g] = \text{length/time}^2, [a] = \text{length} \Rightarrow [g/a] = \text{time}^{-2} \Rightarrow [\sqrt{g/a}] = \text{time}^{-1}.\)\footnote{Remember that \(b\) is not the same in the two instances.\footnote{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.}}
11.1. SAMPLE PROBLEMS

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 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

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

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: \(^5\) adding the three equations yields that

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

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 \( \overline{A} = A/A_0, \) \( \overline{B} = B/A_0, \) and \( \overline{C} = C/A_0. \) Next, dividing the equations by \( k_A A_0 \) and introducing the dimensionless quantities \( \overline{t} = k_A t \) and \( \overline{k} = k_B/k_A, \) we conclude that

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

with the initial conditions \( \overline{A}(0) = 1, \overline{B}(0) = 0, \) and \( \overline{C}(0) = 0, \) where we now regard the dependent variables as functions of the dimensionless time \( \overline{t}. \) In short, we discover that this problem possesses only one “real” parameter \( \overline{k}. \) 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

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

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 \( N \) 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 \( N_0 \).

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

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

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

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

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

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

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

\[
\frac{d^2x}{dt^2} = -\frac{\tau^2 k}{m} x - \frac{\tau b}{m} \frac{dx}{dt} + \tau^2 \frac{F(\tau t)}{m \ell} \tag{11.26}
\]

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 \( \tau \) \textit{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

\[
\frac{d^2x}{dt^2} = -x - \frac{b}{\sqrt{mk}} \frac{dx}{dt} + \frac{F(\tau t)}{k \ell} \quad \Rightarrow \quad \frac{d^2x}{dt^2} = -x - \beta \frac{dx}{dt} + F(\tau) \tag{11.27}
\]

where, in the final form, we have set \( \beta = b/\sqrt{mk} \) and \( F(\tau) = F(\tau t) / k \ell \). The single, dimensionless parameter \( \beta \) contains the essential influence of the three parameters \( m \), \( b \), and \( k \) once differences attributable to scaling have been removed. The dimensionless force \( F \) 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

\[
\begin{align*}
\bar{x}(0) &= x_0 / \ell \\
\bar{v}(0) &= \frac{dx}{dt}(0) = \frac{d(x/\ell)}{dt(\tau/\ell)} = \frac{v_0}{\ell/\tau}
\end{align*} \tag{11.28}
\]

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 \textit{single} second-order differential equation as a \textit{pair} of first-order equations, either

\[
\begin{align*}
\frac{dx}{dt} &= v \\
\frac{dv}{dt} &= -kx - bv + F(t)
\end{align*} \tag{11.29}
\]

in the original dimensional form, or

\[
\begin{align*}
\frac{d\bar{x}}{d\bar{t}} &= \bar{v} \\
\frac{d\bar{v}}{d\bar{t}} &= -\bar{x} - \beta \bar{v} + F(\bar{t})
\end{align*} \tag{11.30}
\]

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

\[
V(t) - L \frac{di}{dt} - \frac{q}{C} - iR = 0 \implies L \frac{di}{dt} + iR + \frac{q}{C} = V(t) \quad (11.31)
\]

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

\[
i = \frac{dq}{dt} \quad (11.32)
\]

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

\[
q(0) = q_0 ; \quad \frac{dq}{dt}(0) = i(0) = i_0 \quad (11.33)
\]

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

\[
L \frac{d^2q}{dt^2} + R \frac{dq}{dt} + \frac{q}{C} = V(t) \quad (11.34)
\]

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, wherein 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.
11.1. SAMPLE PROBLEMS

Figure 11.5: A system of two coupled objects.

Figure 11.6: An object moving in a central force.

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

$$m \frac{d^2 x_1}{dt^2} = -kx_1 + k'(x_2 - x_1) \quad ; \quad m \frac{d^2 x_2}{dt^2} = -kx_2 - k'(x_2 - x_1)$$
(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 $x_i/a = \overline{x}_i$, introducing a dimensionless time variable $t = \omega t$, where $\omega = \sqrt{k/m}$, and setting $\kappa = k'/k$. The equations then become

$$\frac{d^2 \overline{x}_1}{dt^2} = -\overline{x}_1 + \kappa (\overline{x}_2 - \overline{x}_1) \quad ; \quad \frac{d^2 \overline{x}_2}{dt^2} = -\overline{x}_2 - \kappa (\overline{x}_2 - \overline{x}_1)$$
(11.36)

Initial conditions such as

$$\overline{x}_1(0) = \overline{x}_{10} \quad ; \quad \frac{d\overline{x}_1}{dt}(0) = \overline{v}_{10} \quad ; \quad \overline{x}_2(0) = \overline{x}_{20} \quad ; \quad \frac{d\overline{x}_2}{dt}(0) = \overline{v}_{20}$$
(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 \( r^2 = x^2 + y^2 \), \( r = x \hat{i} + y \hat{j} \), and
\[ \hat{r} = \frac{r}{r} = \frac{x \hat{i} + y \hat{j}}{\sqrt{x^2 + y^2}} \] (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 \emph{fixed} force center, in which case
\[ f(r) = -G \frac{mM}{r^2} = -G \frac{mM}{x^2 + y^2} \] (11.42)
where \( G \) is the universal gravitational constant. With this specific force, the members of Eq. (11.40) become
\[ \frac{d^2 x}{dt^2} = -GM \frac{x}{(x^2 + y^2)^{3/2}} ; \quad \frac{d^2 y}{dt^2} = -GM \frac{y}{(x^2 + y^2)^{3/2}} \] (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
\[ \mathbf{x} = \frac{x}{\ell} ; \quad \mathbf{y} = \frac{y}{\ell} ; \quad \mathbf{r} = \frac{r}{\ell} \] (11.44)
With this change, Eq. (11.43) becomes
\[ \frac{d^2 \mathbf{x}}{dt^2} = -\frac{GM}{\ell^3} \frac{\mathbf{x}}{(\mathbf{x}^2 + \mathbf{y}^2)^{3/2}} ; \quad \frac{d^2 \mathbf{y}}{dt^2} = -\frac{GM}{\ell^3} \frac{\mathbf{y}}{(\mathbf{x}^2 + \mathbf{y}^2)^{3/2}} \] (11.45)
Finally, we introduce the dimensionless time variable \( \tilde{t} = t \sqrt{GM/\ell^3} \) to find that
\[ \frac{d^2 \mathbf{x}}{d\tilde{t}^2} = -\frac{\mathbf{x}}{(\mathbf{x}^2 + \mathbf{y}^2)^{3/2}} ; \quad \frac{d^2 \mathbf{y}}{d\tilde{t}^2} = -\frac{\mathbf{y}}{(\mathbf{x}^2 + \mathbf{y}^2)^{3/2}} \] (11.46)
Interestingly, in Cartesian coordinates, all parameters have disappeared from the equations.

\(^9\)Remember, 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

\[ \frac{d\bar{x}}{dt}(0) = \frac{d(x/\ell)}{d(t\sqrt{GM/\ell^3})}(0) = \frac{v_x}{\sqrt{GM/\ell}} ; \quad \frac{d\bar{y}}{dt}(0) = \frac{d(y/\ell)}{d(t\sqrt{GM/\ell^3})}(0) = \frac{v_y}{\sqrt{GM/\ell}} \] (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 \( r_{\text{circ}} \) from the sun with a velocity of magnitude \( v_{\text{circ}} \) 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 \( mv_{\text{circ}}^2/r_{\text{circ}} \) needed for circular motion is exactly provided by the gravitational attraction \( GMm/r_{\text{circ}}^2 \) of the sun. That is, the orbit will be circular if

\[ \frac{mv_{\text{circ}}^2}{r_{\text{circ}}} = \frac{GMm}{r_{\text{circ}}^2} \quad \Rightarrow \quad v_{\text{circ}}^2 = \frac{GM}{r_{\text{circ}}} \] (11.49)

Translated into dimensionless form, this special relationship becomes

\[ \left( \sqrt{\frac{GM}{\ell}} v_{\text{circ}} \right)^2 = \frac{GM}{\ell r_{\text{circ}}} \quad \Rightarrow \quad v_{\text{circ}}^2 = \frac{1}{r_{\text{circ}}} \] (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

\[ E = \frac{1}{2} m(v_x^2 + v_y^2) - \frac{GMm}{\sqrt{x^2 + y^2}} = \text{constant} \] (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

\[ L = m(xv_y - yv_x) = \text{constant} \] (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

\[ \frac{1}{2} (\bar{v}_x^2 + \bar{v}_y^2) - \frac{1}{\sqrt{\bar{x}^2 + \bar{y}^2}} = \text{constant} ; \quad \bar{x}\bar{v}_y - \bar{y}\bar{v}_x = \text{constant} \] (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

\[
u(x, t) = f(x) \cos(\omega t)
\]

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

\[
\frac{d^2 f}{dx^2} + k^2 f = 0
\]

where \( k^2 = \frac{\omega^2}{c^2} \). 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

\[
f(0) = 0 \quad ; \quad f(\ell) = 0
\]

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

\[
-\frac{\hbar^2}{2m} \frac{d^2 \psi}{dx^2} + V(x) \psi = E \psi \quad \Rightarrow \quad \frac{d^2 \psi}{dx^2} + \frac{2m}{\hbar^2} (E - V(x)) \psi = 0
\]

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

\[
\frac{d^2 \psi}{dx^2} + \frac{2mE}{\hbar^2} \psi = 0 \quad ; \quad \psi(0) = \psi(\ell) = 0
\]

Especially if we substitute the shorthand \( k^2 = \frac{2mE}{\hbar^2} \) (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

\[
\frac{d^2 \psi}{dx^2} + \frac{2m}{\hbar^2} \left( E - \frac{1}{2}kx^2 \right) \psi = 0
\]

or, in dimensionless form,

\[
\frac{d^2 \psi}{dx^2} + (2\epsilon - \pi^2) \psi = 0
\]

where \( \omega = \sqrt{k/m} \), \( \epsilon = E/\hbar \omega \) and \( \pi = x\sqrt{\hbar/m\omega} \). 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
\[
\frac{d^2 \psi}{dx^2} + (2 \epsilon - x^2) \psi = 0 ; \quad \psi(0) = 1.0 ; \quad \frac{d\psi}{dx}(0) = 0 ; \quad \lim_{x \to \infty} \psi(x) = 0
\] (11.62)
and yielding even solutions and the other defined by the equation and initial values
\[
\frac{d^2 \psi}{dx^2} + (2 \epsilon - x^2) \psi = 0 ; \quad \psi(0) = 0 ; \quad \frac{d\psi}{dx}(0) = 1 ; \quad \lim_{x \to \infty} \psi(x) = 0
\] (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\(^{12}\) for $\tilde{z}(s)$, finding that
\[
\tilde{z}(s) = \frac{z_0}{s} + \frac{v_{z0} - g}{s^2(s + b/m)}
\] (11.66)

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

---

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

\(^{11}\)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.

\(^{12}\)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.

\(^{13}\)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

\[ \tilde{z}(s) = \frac{z_0}{s} - \frac{mg/b}{s^2} + \frac{m}{b} \left( v_0 + \frac{mg}{b} \right) \left( \frac{1}{s} - \frac{1}{s + b/m} \right) \]  

(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

\[ z(t) = z_0 - \frac{mg}{b} t + \frac{m}{b} \left( v_0 + \frac{mg}{b} \right) \left( 1 - e^{-bt/m} \right) \]  

(11.68)

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

11.4 Solving ODEs Symbolically with MAPLE

Fundamentally, solving an ordinary differential equation involves specifying the equation, finding a solution containing the appropriate number of arbitrary constants, and imposing the appropriate initial or boundary conditions. A few features of the command \texttt{dsolve}, which is the primary ODE-solving command in MAPLE, were discussed briefly in Section 7.8.9. In that section, we used \texttt{dsolve} only for solving a \textit{single} ODE, and we learned that—to that end—the command has two forms. In the first form,

\texttt{dsolve( ODE, DVar(IVar) )}

the command returns the solution of the single ordinary differential equation \textit{ODE} for the dependent variable \textit{DVar} as a function of the single independent variable \textit{IVar}. In this form, \texttt{dsolve} does not impose initial conditions—they haven’t even been specified. Instead, it uses the symbols \texttt{C1, C2, ...} (however many are needed) for the integration constants. These constants must then be determined by subsequent imposition of appropriate initial conditions.\textsuperscript{14}

In the second form, the command \texttt{dsolve} accepts as its first argument a \textit{set} which contains both the differential equation to be solved \textit{and} an appropriate number of initial conditions. The form of the statement would then be

\texttt{dsolve( \{ ODE, IC1, IC2, ...\}, DVar(IVar) )}

This form returns a solution on which the initial conditions \textit{IC1, IC2, ...} have been imposed and which therefore contains no undetermined constants. If, for example, the function to be found is \( y(t) \), then an initial condition on the function itself would have the form \( y(0) = y_0 \) while an initial condition on its first derivative would have the form \( D(y)(0) = v_0 \), where \( y_0 \) and \( v_0 \) are the desired values—and may be symbolic or explicitly numeric.

The command \texttt{dsolve} is also capable of solving (some) \textit{systems} of ordinary differential equations. In this case, the first argument will be a \textit{set} containing all of the differential equations as well as the initial conditions to be imposed, and the second argument will be a \textit{set} containing the several dependent variables. In brief, the statement invoking \texttt{dsolve} will have the form

\texttt{dsolve( \{ ODE1, ODE2, ..., IC1, IC2, ...\}, \{ DVar1(IVar), DVar2(IVar), ...\} )}

\textsuperscript{14}When only one function in the given ODE is differentiated, MAPLE can determine that function for itself and the second argument can be omitted.
In some cases, arguments to \texttt{dsolve} beyond those described above can be used to exert explicit control over the specific methods tried, the form of the output produced, or other behaviors of the basic command. Details on these embellishments are described in the help message accessed with the command \texttt{?dsolve} and, for options more specific to particular contexts, with the commands \texttt{?dsolve,ICs}, \texttt{?dsolve,systems}, \texttt{?dsolve,series}, \texttt{?dsolve,numeric}, and \texttt{?dsolve,integral_transforms}.

As we learned in Section 7.8.9, we can request MAPLE to tell us a bit about what it is doing by setting a particular infolevel away from its default value to the value 3 with the statement

\begin{verbatim}
infolevel[dsolve] := 3;
\end{verbatim}

Once this value has been established, MAPLE will provide a running report of what it is doing as it seeks a solution to an ODE with \texttt{dsolve}.\footnote{Note that, since newer versions of MAPLE may invoke improved approaches, the reports produced by different versions of MAPLE may not be identical.}

Finally, you are urged to look at the additional MAPLE commands contained in the packages \texttt{DEtools} and \texttt{PDEtools}. These packages contain tools for plotting solutions (including Poincare sections), changing variables, finding indicial equations, finding integrating factors, and performing numerous other tasks. In particular, should you desire information before attempting a solution, you could invoke the command \texttt{odeadvisor},\footnote{See the on-line message produced by the command \texttt{?odeadvisor}.} which in essence analyzes the ODE submitted as its argument and provides hints/suggestions for pursuing its solution. This command is a component of the package \texttt{DEtools}, and will be available only after the statement \texttt{with(DEtools)} has been executed.\footnote{See Section 7.9.}

### 11.4.1 Projectile in a Viscous Medium

For the sake of a comparison with the result obtained in Section 11.2, we begin by using MAPLE to solve the third member of Eq. (11.4). The necessary commands are\footnote{As in Chapter 7, we abbreviate the presentation of MAPLE dialogs by making liberal use of terminating colons to suppress intermediate output. You are urged to duplicate the dialog in an actual session with MAPLE, replacing all terminating colons with semi-colons.} \footnote{See Section 7.8.9 for a discussion of the use of the inert command \texttt{Diff} in the present context.}

\begin{verbatim}
> infolevel[dsolve] := 3: Request details of what MAPLE is doing.
> eq := m*Diff(z(t),t$2)=-m*g-b*Diff(z(t),t): Define the differential equation.
> soln := dsolve( eq, z(t) ); Solve the equation for z as a function of t.
\end{verbatim}

Methods for second order ODEs:
--- Trying classification methods ---
trying a quadrature
trying high order exact linear fully integrable
-> Calling odsolve with the ODE diff(_b(_a) _a) = -(m*g+b*_b(_a))/m _b(_a)

*** Sublevel 2 ***
Methods for first order ODEs:
--- Trying classification methods ---
trying a quadrature
trying 1st order linear
<- 1st order linear successful
<- high order exact linear fully integrable successful

\begin{verbatim}
soln := z(t) = -C1m \frac{e^{-bt/m}}{b} - \frac{gmt}{b} + C2
\end{verbatim}
Note that MAPLE has here assumed that $b \neq 0$. More careful treatment would be required if $b = 0$, i.e., in the case of no damping.

To complete the solution, we must of course impose appropriate initial conditions. Thus, we continue with the statements

```maple
> z := rhs(soln):
> vz := diff(z, t):
> eq1 := eval(z, t=0) = z0:
> eq2 := eval(vz, t=0) = vz0:
> tmp := solve({eq1, eq2}, {_C1, _C2}):
> z := eval(z, tmp):
> z := collect(z, z0);
```

Find position as an expression.

```maple
Find velocity as an expression.
```

Impose initial condition on position.

Impose initial condition on velocity.

Solve for integration constants.

Substitute constants into solution.

Recast result in a slightly better form.

```maple
z := z0 - \frac{me^{-bt/m}(mg + vz0b)}{b^2} - \frac{mgt}{b} + \frac{m^2g + mvz0b}{b^2}
```

Evaluate velocity again.

```maple
vz := \frac{e^{-bt/m}(mg + vz0b)}{b} - \frac{mg}{b}
```

The clearly equivalent forms

$$z(t) = z_0 - \frac{mg}{b} t + \frac{m}{b} \left( vz_0 + \frac{mg}{b} \right) \left( 1 - e^{-bt/m} \right)$$

(11.69)

and

$$v_z(t) = -\frac{mg}{b} + \left( vz_0 + \frac{mg}{b} \right) e^{-bt/m}$$

(11.70)

for the position and velocity are even better.\(^{20}\) This result for $z(t)$ agrees fully with the result obtained in Eq. (11.68).

Wisdom suggests that we should whenever possible actually verify any solution that MAPLE generates. To this end, we substitute the solution into the original equation, evaluate the derivatives explicitly, and simplify the result with the statements

```maple
> value(eval(eq, z(t)=z));
> tmp := simplify( % );
```

```maple
tmp := -(mg + vz0b) e^{-bt/m} = -(mg + vz0b) e^{-bt/m}
```

The left and right sides of this result are clearly equal, though we could make that more explicit either by using the statement

```maple
> lhs(tmp) - rhs(tmp);
0
```

or by invoking the test

```maple
> is( tmp );
true
```

where, with `tmp` having the form $a = b$, the command `is` simply returns `true` or `false` depending on whether the specified expression is, in fact, true or false. To complete the verification, we check satisfaction of the initial conditions with the statement

\(^{20}\)Try to find a way to persuade MAPLE to recast the immediate output into these forms.
All is indeed well.

One further check on this solution involves examining its limit as $b$—the viscous damping—becomes small. For the coordinate and velocity, we find respectively that

$$z_0 + v_z t - \frac{1}{2} gt^2 - \frac{bt^2}{2m} \left( v_{z0} - \frac{1}{3} gt \right) + O(b^2)$$

The terms free of $b$ (i.e., terms in $b^0 = 1$) clearly agree with the known results for free fall in the absence of viscous resistance.

The solution of this problem when motion occurs in two or three of the coordinate directions is explored in one of the exercises.

We have, of course, taken a bit longer route than necessary so as to illustrate the use of MAPLE to find the undetermined constants in the general solution to an ODE. We could combine the first several of our statements into an invocation of `dsolve` that includes the incorporation of the initial conditions with the statements

```
> restart:
> eq := m*Diff(z(t),t$2)=-m*g-b*Diff(z(t),t):
> soln := dsolve( {eq, z(0)=z0, D(z)(0)=vz0}, z(t) );
```

These statements lead us in rather fewer steps to the result already presented in Eq. (11.69) though not immediately in the form there presented.

### 11.4.2 Logistic Growth

Consider next Eq. (11.22) governing the logistic growth of a population $N$ in an environment with carrying capacity $N_c$. We seek the solution with the MAPLE statements

```
> eq := Diff( N(t),t ) = k*N(t)*(1-N(t)/Nc):
> soln := dsolve( {eq, N(0)=N0}, N(t) );
```

We can apply several tests of validity to this result with the statements

---

21 Here, we must use the command `series` rather than the command `taylor` because—at least superficially—the series in $b$ appears to have terms with negative powers of $b$.

22 Again, we have taken the liberty to rearrange the terms in MAPLE’s output to facilitate recognition of the limits. We shall continue to take this liberty.
> simplify( eval(soln, t=0) );
\[ N(0) = N_0 \] Verify satisfaction of initial condition.

> value( eval( eq, N(t) = rhs(soln) ) );
> is( % );
\[ N(0) = N_0 \] Verify satisfaction of ODE.

\[ \text{true} \]

> assume( k > 0 );
Examine limit as \( t \to \infty \). As expected, the solution approaches the carrying capacity, regardless of the initial population.

> limit( soln, t=infinity );
\[ \lim_{t \to \infty} N(t) = N_c \]
Remove assumptions from \( k \).

The solution passes all three tests.

Let us conclude the discussion of this example by displaying the evolution of the population in a graph. We begin by casting the expression to be plotted in a dimensionless form, choosing \( N_c \) as the unit in terms of which to express the population.\(^{23}\) We extract the function to be plotted and recast it with the statements

\[ N := \text{rhs( soln )/Nc}; \]
Recast to measure population \( N \) in units of \( N_c \).

\[ N := \frac{N_0}{-N_0 - e^{-kt}N_c + e^{-kt}N_0} \]

\[ N := \text{simplify( subs( N0=alpha*Nc, \% ) )}; \]
Recast to measure initial population in units of \( N_c \).

\[ N := \frac{-\alpha - e^{-kt} + e^{-kt}\alpha}{\alpha} \]

Were we doing this calculation by hand, we might swallow the minus sign out front by changing the sign of each term in the denominator, divide numerator and denominator by \( \alpha \), and then collect the terms in the denominator involving \( e^{-kt} \) to yield the result

\[ N(t) = \frac{1}{1 + \frac{e^{-kt}(1 - \alpha)}{\alpha}} \] (11.71)

We can achieve that recasting with the statements

\[ \text{top} := \text{numer}(N); \]
Extract numerator of \( N \).

\[ \text{bot} := \text{denom}(N); \]
Extract denominator of \( N \).

\[ N := 1/\text{collect}( \text{bot/top, exp(-k*t))}; \]
Recompute \( N \).

\[ N := \frac{1}{1 + e^{-\tau}(1 - \alpha)/\alpha} \]

Finally, we make one further recasting to measure time in units of \( 1/k \) with the statement

\[ N := \text{subs( t=tau/k, N )}; \]

\[ N := \frac{1}{1 + e^{-\tau}(1 - \alpha)/\alpha} \]

With these rescalings, we have but one parameter [physically (biologically?) the initial population measured in units of the carrying capacity]. We produce the graph in Fig. 11.7 with the statements

\[ N1 := \text{eval( N, alpha=0.25 )}; \]
\[ N2 := \text{eval( N, alpha=0.5 )}; \]

\(^{23}\)Since different versions of MAPLE present results in equivalent but distinct arrangements, you may need different statements than here illustrated to arrive at the desired end result.
11.4. SOLVING ODES SYMBOLICALLY WITH MAPLE

Figure 11.7: Logistic growth or decay of a population when, starting with the highest graph, the initial population is 4.0, 2.0, 1.0, 0.5, and 0.25 times the carrying capacity of the environment.

Regardless of the initial population, all solutions converge monotonically on the carrying capacity (1.0 in the units we are using). Further, if we execute the statements

```maple
> N3 := eval( N, alpha=1.0 ):
> N4 := eval( N, alpha=2.0 ):
> N5 := eval( N, alpha=4.0 ):
> plot( [N1,N2,N3,N4,N5], tau=0.0..5.0, y=0.0..4.0, color=black,
        thickness=3, linestyle=1, labels=["tau","N/Nc"],
        title="Logistic Behavior", labelfont=[TIMES,ROMAN,16],
        titlefont=[TIMES,ROMAN,20], axesfont=[TIMES,ROMAN,14],
        labeldirections=[HORIZONTAL,VERTICAL] );
```

we generate the graph in Fig. 11.8, which shows the initial exponential growth when the population is well below the carrying capacity but then reveals the leveling off as the population approaches the carrying capacity.

11.4.3 Damped Harmonic Oscillator

A third example of the use of MAPLE to solve a single second-order, linear, constant-coefficient, homogeneous, ordinary differential equation was presented in Chapter 7. You are invited to review the discussion of the damped harmonic oscillator in Sections 7.8.9 and 11.1.4.
11.4.4 Chain Radioactive Decay

As a first example of the use of MAPLE to solve a system of linear equations, let us determine the behavior of the radioactive decay chain described by Eq. (11.19). For simplicity, we suppose the initial values \( A(0) = A_0 \) and \( B(0) = C(0) = 0 \), i.e., we start with some \( A \) and no \( B \) or \( C \). The temporal evolution of this system is found with the MAPLE statements

\[
\begin{align*}
> & \text{eqA := Diff( A(t), t ) = -kA*A(t):} \quad \text{Enter equations, suppressing display.} \\
> & \text{eqB := Diff( B(t), t ) = kA*A(t)-kB*B(t):} \\
> & \text{eqC := Diff( C(t), t ) = kB*B(t):} \\
> & \text{soln := simplify( dsolve( \{eqA, eqB, eqC,} \\
> & \hspace{1cm} A(0)=A0, B(0)=0, C(0)=0\},} \\
> & \hspace{1cm} \{A(t),B(t),C(t)\} )); \\
> \text{MAPLE displays the result}\;
\end{align*}
\]

\[
\begin{align*}
soln := \left\{ A(t) &= A_0 e^{-kA t}, \\
B(t) &= -\frac{A_0 k_A (e^{-kA t} - e^{-kB t})}{k_A - k_B}, \\
C(t) &= \frac{A_0 (k_A e^{-kB t} + kB e^{-kA t} - k_B)}{k_A - k_B}\right\}
\end{align*}
\]

Evidently, in this chain decay, we see a linear combination of two exponential decays, each with its own distinct half-life.

We conclude the discussion of this example by generating a quick graph of \( A(t) \), \( B(t) \), and \( C(t) \) for a specific set of values. We prepare for the graph with the statements

\[
\begin{align*}
> & A := \text{rhs(soln[1]):} \quad \text{Extract solutions and anticipate we will set} \\
> & B := \text{limit(rhs(soln[2]), kA=kB ):} \quad k_A = k_B. \ (\text{Expressions are indeterminate.}) \\
> & C := \text{limit(rhs(soln[3]), kA=kB ):} \\
> & A0 := 1000: kA:=0.1 : kB := 0.1: \quad \text{Set parameters.} \\
\end{align*}
\]

Then, we produce the graph with the single statement

---

24We have rearranged MAPLE’s actual output for clarity of expression.
Figure 11.9: Radioactive decay of A, B, and C. The graph starting at 1000 is A; the graph rising to 1000 at \( t = 50.0 \) is C; and the remaining graph is B.

\[
\text{plot( \{A,B,C\}, t=0.0..50.0, y=0.0..1000.0, color=black, thickness=3, linestyle=1, labels=\"t\",\"A,B,C\"), labelsfont=[TIMES,ROMAN,16], axesfont=[TIMES,ROMAN,14], labeldirections=[HORIZONTAL,VERTICAL] );}
\]

The resulting graph is shown in Fig. 11.9.

### 11.4.5 Coupled Oscillators

Let us next solve for the motion of the coupled oscillators described in Section 11.1.6. We shall address the problem in the dimensionless form presented in Eq. (11.36), imposing the general initial conditions in Eq. (11.37). For simplicity, however, we will take the initial velocities both to be zero.\(^{25}\)

First, suppressing the display of MAPLE's response, we enter the equations with the statements\(^ {26}\)

\[
> \text{eq1 := Diff(x1(t),t$2) + x1(t) - kappa*(x2(t)-x1(t)) = 0:}
> \text{eq2 := Diff(x2(t),t$2) + x2(t) + kappa*(x2(t)-x1(t)) = 0:}
\]

Then we create a set containing the initial values with the statement

\[
> \text{ics := \{ x1(0)=x10, x2(0)=x20, D(x1)(0)=0, D(x2)(0)=0 \}:}
\]

and, finally, we request the solution with the statement

\[
> \text{soln := dsolve( \{eq1,eq2\} union ics, \{x1(t), x2(t)\} );}
\]

\[
\text{soln := } \left\{ x_2(t) = \frac{1}{2}(x_{10} + x_{20}) \cos(t) + \frac{1}{2}(-x_{10} + x_{20}) \cos(\sqrt{2\kappa + 1} \ t), \right. \\
\quad \left. x_1(t) = -\frac{1}{2}(-x_{10} + x_{20}) \cos(\sqrt{2\kappa + 1} \ t) + \frac{1}{2}(x_{10} + x_{20}) \cos(t) \right\}
\]

\(^{25}\)You are urged to explore other initial conditions.

\(^{26}\)In the derivation of the dimensionless equations, we used overbars to identify the dimensionless quantities. We here drop that refinement in the notation, omitting the overbars but understanding that the quantities represented are still dimensionless.
Clearly, the solution is a superposition of different sinusoidal oscillations, one at frequency 1 (in units of $\omega = \sqrt{k/m}$) and the other at frequency $\sqrt{2\kappa + 1}$ (again in units of $\omega$). Clearly also, only the term at frequency 1 is present if $x_{20} = x_{10}$ (the lower-frequency normal mode, in which the two objects oscillate with equal amplitude and in phase), and only the term at frequency $\sqrt{2\kappa + 1}$ is present if $x_{20} = -x_{10}$ (the higher-frequency normal mode, in which the objects oscillate with equal amplitude but out of phase).

### 11.4.6 Standing Waves in a String

Boundary value problems need to be treated differently from initial value problems. To find the standing waves in a string that satisfy Eqs. (11.56) and (11.57), for example, we might (naively, as it turns out) try the statements:

```
> eq := Diff( f(x), x$2 ) + k^2 * f(x) = 0:
> soln := dsolve( {eq,f(0)=0,f(l)=0}, f(x) );
```

Enter equation.

Solve equation for $f$ as function of $x$ subject to zero boundary conditions at each end of the string.

MAPLE returns $f(x) = 0$, having concluded that only the trivial solution to the ODE can be made to satisfy both boundary conditions. In imposing the boundary conditions, MAPLE doesn’t recognize that we can find non-trivial solutions, but only if we are prepared to constrain $k$ to a limited set of values.\(^{27}\)

Evidently, we must be more careful. To help MAPLE recognize that the problem is a bit more subtle, we simply solve the differential equation without specifying the boundary conditions, thereby generating a solution that includes two initially undetermined integration constants. Then, we impose the boundary conditions more deliberately. In steps, we solve the equation with the statement:\(^{28}\)

```
> soln := dsolve( eq, f(x) );
```

Next, we determine the consequences of the boundary conditions with the statements:

```
> eq1 := eval( rhs(soln), x=0 ) = 0;  
  eq1 := _C2 = 0
> eq2 := eval( rhs(soln), x=l ) = 0;  
  eq2 := _C1 sin(kl) + _C2 cos(kl) = 0
```

At this point, we recognize that we have a pair of equations to be solved simultaneously for $\_C1$ and $\_C2$. We also recognize, however, that these equations are homogeneous and will in fact have non-trivial solutions only if the determinant of the coefficient matrix happens to be zero. To complete the problem, then, we must extract that coefficient matrix, set its determinant to zero, and solve the resulting equation to find any values of $k$ that may admit non-trivial solutions to the equations themselves. We accomplish that end with the statements\(^{29}\)

\(^{27}\)Had we imposed different values on the solution at the two ends of the interval, a non-trivial solution would have emerged. Try it, say, with $f(0) = 0$ and $f(\ell) = x_1$.

\(^{28}\)Different implementations of MAPLE on different platforms may exchange the roles of $\_C1$ and $\_C2$.

\(^{29}\)Note that different versions of MAPLE may exchange the role of the special symbols $\_C1$ and $\_C2$.  

11.4. SOLVING ODES SYMBOLICALLY WITH MAPLE

\[ (M,b) := \text{LinearAlgebra[GenerateMatrix]}( [eq1,eq2],[_C1,_C2] ); \]

\[
(M,b) := \begin{bmatrix} 0 & 1 \\ \sin(k\ell) & \cos(k\ell) \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \end{bmatrix}
\]

\[ \text{LinearAlgebra[Determinant]}( M ); \]

\[-\sin(k\ell)\]

\>_\text{EnvAllSolutions} := 'true'; \]

\[ rts := \text{solve}( \% = 0, k ); \]

\[ rts := \frac{\pi Z_1\ell}{\ell} \]

Here \( Z_1 \) —a symbol MAPLE has introduced—stands for an arbitrary integer; we have discovered that, with appropriate setting of an environment variable, MAPLE’s command \text{solve} can even handle some equations involving trigonometric functions.

Having now found acceptable values of \( k \), we find the corresponding values of \(_C1\) and \(_C2\) by returning to equations \text{eq1} and \text{eq2}. For the allowed values of \( k \),

\[ \text{eval( eq1, k=rts );} \]

\[ _C2 = 0 \]

\[ \text{eval( eq2, k=rts );} \]

\[ _C2 (-1)^{Z_1} = 0 \]

The first of these equations tells us that we must choose \(_C2 = 0\)—a fact we really knew at the very beginning. Once we have done that, however, the second equation is automatically satisfied, and we learn nothing about \(_C1\), which evidently remains arbitrary insofar as the conditions of the problem are concerned.

In the end, we discover that we can find acceptable solutions to the original boundary value problem only when \( k \) has one of the values \( n\pi/\ell \) (for a cleaner appearance, we replace \( Z_1 \) with \( n \)) and \(_C2\) is set to zero. Within MAPLE, we find that

\[ \text{eval( soln, \{k=rts, eq1\});} \]

\[ \text{subs( _Z1=n, \%);} \]

\[ f(x) = _C1 \sin \frac{n\pi x}{\ell} \]

where \( n \) assumes any of the values 1, 2, 3, … (but not the value 0—because the solution reduces to zero in that case—and not negative values—because they really change only the overall sign of the solution and could be seen as simply an alternative choice of \(_C1\)). Further, since \( \omega = kc \), we would associate with this solution the frequency

\[ \omega_n = k_n c = n \frac{\pi c}{\ell} = n \omega_1 \]

where \( \omega_1 \) is the fundamental frequency. We find, in particular, that all allowed frequencies are integer multiples of the fundamental frequency.

11.4.7 Infinite Depth Quantum Well

As pointed out in Section 11.1.9, the quantum problem of a particle in an infinitely deep, one-dimensional, square potential well is mathematically identical to the problem we have just addressed. Since the allowed values of \( k \) have turned out to be \( n\pi/\ell \), we then conclude that the allowed energies in the quantum problem would be

\[ E_n = \frac{\hbar^2 k_n^2}{2m} = \frac{n^2 \hbar^2 \pi^2}{2m \ell^2} \]
Further, we would associate the (unnormalized) wave function
\[ \psi_n(x) = N_n \sin \frac{n \pi x}{\ell} \]  
with the energy \( E_n \). In the quantum problem, the allowed energies are not integer multiples of the lowest energy; rather, the allowed energies increase in proportion to the squares of the integers.

### 11.4.8 Finding Solutions by Series Methods

Particularly when analytic methods appear to be yielding little result, we are sometimes tempted to invoke the method of Frobenius, representing the solution to the ODE of interest as a power series of the form
\[ f(x) = \sum_{n=0}^{\infty} a_n x^{n+\gamma} \]
and seeking an indicial equation determining \( \gamma \) and recursion relationships determining \( a_n \) from a few of the early coefficients. In progressing through its succession of methods, MAPLE will in some cases automatically adopt this method as a last resort. If we know ahead of time that we want a solution generated with this method (or some other particular method), we can circumvent MAPLE’s trial of other methods and specify the desired method by stipulating an optional argument to the command \texttt{dsolve}. The statement
\[
> \text{dsolve( \{ ODE, ICs\}, DVar(IVar), 'type=series' )}
\]
will cause \texttt{dsolve} to abort its normal search and jump directly to the specified method.\(^{30}\) The series is worked out to an order controlled by the MAPLE environment variable \texttt{Order}, whose default value is 6.

Details on the generation of series solutions with MAPLE can be found in the MAPLE manuals, and particularly in the message displayed in response to the command \texttt{?dsolve,series}. We illustrate with a simple example, specifically, solution of the equation
\[ \frac{d^2 x}{dt^2} + \omega^2 x(t) = 0; \quad x(0) = x_0; \quad \frac{dx}{dt}(0) = v_0 \]  
(11.76)
We invoke the statements
\[
> \text{Order := 8;}
> \text{eq := Diff(x(t), t$2$) + \omega^2 x(t) = 0;}
> \text{soln := dsolve( \{eq, x(0)=x0, D(x)(0)=v0\}, x(t), 'type=series' );}
> \text{soln := convert( soln, polynom );}
> \text{soln := collect( soln, [x0, v0] );}
\]
\[ \text{soln := } x(t) = \left( 1 - \frac{1}{2} \omega^2 t^2 + \frac{1}{24} \omega^4 t^4 - \frac{1}{720} \omega^6 t^6 \right) x_0 + \left( t - \frac{1}{6} \omega^2 t^3 + \frac{1}{120} \omega^4 t^5 - \frac{1}{5040} \omega^6 t^7 \right) v_0 \]
The statements
\[
> \text{convert( series( cos(\omega*t), t, 8 ), polynom );}
> \left( 1 - \frac{1}{2} \omega^2 t^2 + \frac{1}{24} \omega^4 t^4 - \frac{1}{720} \omega^6 t^6 \right)
> \text{convert( series( sin(\omega*t)/\omega, t, 8 ), polynom );}
> \left( t - \frac{1}{6} \omega^2 t^3 + \frac{1}{120} \omega^4 t^5 - \frac{1}{5040} \omega^6 t^7 \right)
\]
\(^{30}\)Other possibilities for this method are listed in the MAPLE manuals, in particular in the message displayed in response to the command \texttt{?dsolve}. 

provide evidence that the solution obtained as a series is, more generally

\[ x(t) = x_0 \cos(\omega t) + \frac{v_0}{\omega} \sin(\omega t) \]  

(11.77)

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 independent 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

\[ \frac{dx}{dt} = f(x, t) ; \quad x(0) = x_0 \]  

(11.78)

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,\(^{31}\) we can write the approximation

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

(11.79)

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

\[ f_{\text{old}} = f(x_{\text{old}}, t_{\text{old}}) ; \quad x_{\text{new}} = x_{\text{old}} + f_{\text{old}} \Delta t \quad ; \quad t_{\text{new}} = t_{\text{old}} + \Delta t \]  

(11.80)

Starting from the initial condition as the first \( x_{\text{old}} \) (and a choice of time step \( \Delta t \)—see later), we can then use these stepping equations repeatedly to move from knowledge of \( x(0) \) to knowledge of \( x(\Delta t) \) to knowledge of \( x(2\Delta t) \) to \ldots, continuing as long as our patience endures. The basic strategy of Euler’s method would be summarized in the algorithm in Table 11.1.\(^{32}\) As this example shows, we can generate a complete—though approximate—solution from initial knowledge of only

\(^{31}\)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.

\(^{32}\)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.
Table 11.1: Simple Euler algorithm.

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$dt \leftarrow \Delta t$</td>
<td>Choose time step.</td>
</tr>
<tr>
<td>$t_{old} \leftarrow 0.0$</td>
<td>Initialize $t_{old}$.</td>
</tr>
<tr>
<td>$x_{old} \leftarrow x_0$</td>
<td>Initialize $x_{old}$.</td>
</tr>
<tr>
<td><strong>loop</strong></td>
<td></td>
</tr>
<tr>
<td>$f_{old} \leftarrow f(x_{old}, t_{old})$</td>
<td>Evaluate $f_{old}$.</td>
</tr>
<tr>
<td>print, $t_{old}$, $x_{old}$, $f_{old}$</td>
<td>Display results.</td>
</tr>
<tr>
<td>exit_loop when done</td>
<td></td>
</tr>
<tr>
<td>$x_{new} \leftarrow x_{old} + f_{old} \cdot dt$</td>
<td>Calculate values at new time.</td>
</tr>
<tr>
<td>$t_{new} \leftarrow t_{old} + dt$</td>
<td></td>
</tr>
<tr>
<td>$x_{old} \leftarrow x_{new}$</td>
<td>Replace old values with new.</td>
</tr>
<tr>
<td>$t_{old} \leftarrow t_{new}$</td>
<td></td>
</tr>
<tr>
<td><strong>end_loop</strong></td>
<td></td>
</tr>
</tbody>
</table>

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 $\Delta t$.

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

\[
A(t + \Delta t) = A(t) + \frac{dA}{dt}(t) \Delta t = A(t) - k_A A(t) \Delta t \quad (11.81)
\]

\[
B(t + \Delta t) = B(t) + \left[ k_A A(t) - k_B B(t) \right] \Delta t \quad (11.82)
\]

\[
C(t + \Delta t) = C(t) + k_B B(t) \Delta t \quad (11.83)
\]

For definiteness, we take $A(0) = 1000.000$, $B(0) = C(0) = 0.000$, and $k_A = k_B = 0.100$, and we select a time step of $\Delta t = 0.250$.\(^{31}\) Equations (11.81), (11.82), and (11.83) at $t = 0.0$ then become

\[
A(0.00 + 0.25) = 1000.0 - 0.1 \cdot 1000.0 \cdot 0.25 = 975.000 = A(0.25)
\]

\[
B(0.00 + 0.25) = 0.0 + \left[ 0.1 \cdot 1000.0 - 0.1 \cdot 0.0 \right] \cdot 0.25 = 25.000 = B(0.25) \quad (11.84)
\]

\[
C(0.00 + 0.25) = 0.0 + 0.1 \cdot 0.0 \cdot 0.25 = 0.000 = C(0.25)
\]

\(^{31}\)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.

\(^{34}\)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.
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>
</tbody>
</table>

Continuing the algorithm further by applying Eqs. (11.81), (11.82), and (11.83) for $t = 0.25$, we find

$$A(0.25 + 0.25) = 975.0 - 0.1 \cdot 975.0 \cdot 0.25 = 950.625 = A(0.50)$$

$$B(0.25 + 0.25) = 25.0 + \left[0.1 \cdot 975.0 - 0.1 \cdot 25.0\right] \cdot 0.25 = 48.750 = B(0.50) \quad (11.85)$$

$$C(0.25 + 0.25) = 0.0 + 0.1 \cdot 25.0 \cdot 0.25 = 0.625 = C(0.50)$$

Going yet one more step, Eqs. (11.81), (11.82), and (11.83) for $t = 0.50$ yield that

$$A(0.50 + 0.25) = 950.625 - 0.1 \cdot 950.625 \cdot 0.25 = 926.859 = A(0.75)$$

$$B(0.50 + 0.25) = 48.75 + \left[0.1 \cdot 950.625 - 0.1 \cdot 48.75\right] \cdot 0.25 = 71.267 = B(0.75) \quad (11.86)$$

$$C(0.50 + 0.25) = 0.625 + 0.1 \cdot 48.75 \cdot 0.25 = 1.844 = C(0.75)$$

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. (11.81), (11.82), and (11.83). 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.\(^{35}\)

### 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 round-off 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 round-off error). The \textit{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 $x_{\text{old}}$ and $t_{\text{old}}$ and a chosen time step $\Delta t$, we

1. calculate $f_{\text{old}}$,

2. calculate the predicted values

$$x_{\text{pred}} = x_{\text{old}} + f_{\text{old}} \Delta t ; \quad t_{\text{new}} = t_{\text{old}} + \Delta t$$

\(^{35}\)Actually, however, that the sum $A + B + C$ preserves its initial value is not really a check on the accuracy of the method in \textit{this example}. If we simply add the three stepping equations we are using [Eqs. (11.81)–(11.83)], we find that $A(t + \Delta t) + B(t + \Delta t) + C(t + \Delta t) = A(t) + B(t) + C(t)$. \textit{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.

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Choose time step.</td>
</tr>
<tr>
<td>2</td>
<td>Initialize ( t_{old} ).</td>
</tr>
<tr>
<td>3</td>
<td>Initialize ( x_{old} ).</td>
</tr>
<tr>
<td>4</td>
<td>Evaluate ( f_{old} ).</td>
</tr>
<tr>
<td>5</td>
<td>Display results.</td>
</tr>
<tr>
<td>6</td>
<td>Calculate predicted value.</td>
</tr>
<tr>
<td>7</td>
<td>Evaluate ( f_{pred} ).</td>
</tr>
<tr>
<td>8</td>
<td>Calculate final (corrected) value.</td>
</tr>
<tr>
<td>9</td>
<td>Replace old values with new.</td>
</tr>
</tbody>
</table>

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

4. calculate the final (corrected) value

\[
x_{new} = x_{old} + \frac{1}{2} \left( f_{old} + f_{pred} \right) \Delta t
\]  

(11.88)

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.\(^{36}\)

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

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

\[
= x(t) + f(x, t) \Delta t + \frac{1}{2} \frac{df(x, t)}{dt} \Delta t^2 + O(\Delta t^3)
\]

\(^{36}\)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.
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>1000.000</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>

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

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

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 $x' = x(t) + b \Delta t f(x(t), t)$, 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. (11.89). 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

\[
x(t + \Delta t) = x(t) + w_1 f(x(t), t) \Delta t
\]

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

\[
= x(t) + (w_1 + w_2) f(x(t), t) \Delta t
\]

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

\[37\text{Note that the steps laid out here reduce to the improved Euler method if we take } a = b = 1 \text{ and } w_1 = w_2 = 1/2.\]
which will match Eq. (11.89) through terms of order $O(\Delta t^2)$ if we choose

$$w_1 + w_2 = 1 \quad aw_2 = bw_2 = \frac{1}{2} \quad 	ext{or} \quad w_1 + w_2 = 1 \quad a = b \quad w_2 = \frac{1}{2a}$$

We have discovered a multitude of Runge-Kutta schemes, one corresponding to each possible choice of $a$, $b$, $w_1$, and $w_2$. One common choice embodies the values in footnote 37—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

$$t' = t + \frac{1}{2} \Delta t \quad x' = x + \frac{1}{2} f(x, t) \Delta t \quad x(t + \Delta t) = x(t) + f(x', t') \Delta t$$

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 $x_{\text{new}}$ in the above algorithm:

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k_1 \leftarrow f(x_{\text{old}}, t_{\text{old}}) \cdot \Delta t$</td>
<td>Find change using rate at beginning.</td>
</tr>
<tr>
<td>$k_2 \leftarrow f(x_{\text{old}} + k_1/2, t_{\text{old}} + \Delta t/2) \cdot \Delta t$</td>
<td>Estimate change using rate at midpoint.</td>
</tr>
<tr>
<td>$k_3 \leftarrow f(x_{\text{old}} + k_2/2, t_{\text{old}} + \Delta t/2) \cdot \Delta t$</td>
<td>Estimate change using refined rate at midpoint.</td>
</tr>
<tr>
<td>$k_4 \leftarrow f(x_{\text{old}} + k_3, t_{\text{old}} + \Delta t) \cdot \Delta t$</td>
<td>Estimate change using rate at end.</td>
</tr>
<tr>
<td>$x_{\text{new}} \leftarrow x_{\text{old}} + (k_1 + 2k_2 + 2k_3 + k_4)/6$</td>
<td>Calculate final value.</td>
</tr>
</tbody>
</table>

38 Actually, we choose only $a$. Then $b = a$, $w_2 = 1/2a$, and $w_1 = 1 - w_2$ are fixed.
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

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

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 value of $\xi$. Nonetheless, we can conclude that

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

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

40 This assertion is refined in the next paragraphs.

41 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.
or that
\[
\left| x_{\text{exact}}(t + \Delta t) - x_{\text{Euler}}(t + \Delta t) \right| \leq \text{Max} \left( \frac{1}{2} \frac{d^2 x}{dt^2}(\xi) \right) \Delta t^2
\] (11.96)

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. (11.95) can be deduced for all of the methods that we have described in this section. Without proof, we present the properties
\[
x_{\text{exact}}(t + \Delta t) = x_{\text{Euler}}(t + \Delta t) + O(\Delta t^2) \quad (11.97)
\]
\[
x_{\text{ImpEul}}(t + \Delta t) + O(\Delta t^3) \quad (11.98)
\]
\[
x_{\text{RK2}}(t + \Delta t) + O(\Delta t^3) \quad (11.99)
\]
\[
x_{\text{RK4}}(t + \Delta t) + O(\Delta t^5) \quad (11.100)
\]

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 \( \Delta t \), \( \Delta t^2 \), \( \Delta t^2 \), 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

---

11.6. ALGORITHMS FOR SOLVING ODES NUMERICALLY

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. (11.78), 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

$$x_{i+1} = x_i + \frac{\Delta t}{24} \left( 55f_i - 59f_{i-1} + 37f_{i-2} - 9f_{i-3} \right) \tag{11.101}$$

(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 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. (11.101), we might use a fourth-order Runge-Kutta method to obtain

---

43 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.

44 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.

$x_1$, $x_2$, 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. (11.101) 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

$$x_{i+1} = x_i + \frac{\Delta t}{24} \left( 9f_{i+1} + 19f_i - 5f_{i-1} + f_{i-2} \right)$$

(11.102)

Once a sufficient number of values is in hand, we could at each step invoke Eq. (11.101) to predict the next solution and then Eq. (11.102) to refine it. We might call the resulting method the Adams-Bashforth-Moulton method.

11.7 Solving ODEs Numerically with IDL

Note: All IDL program (.pro) files referred to in this chapter are available in the directory \$HEAD/idl, where (as defined in the Local Guide) \$HEAD must be replaced by the appropriate path for your site. At some sites, this directory or some other directory containing these files may also have been placed in IDL’s default search path. If so, the files can be found by IDL without explicit specification of a path. Otherwise, you will have to use the full path to copy them into your default directory to access them.

ODEs can be solved numerically either by using IDL’s elementary commands or, more simply, by invoking one of the built-in routines, which include IDL-supplied routines \texttt{rk4} and \texttt{lsode} and Lawrence adapted routines \texttt{ludiffeq.23} and \texttt{ludiffeq.45}.

11.7.1 Using Elementary Commands

To carry the process worked out by hand in Section 11.6.1 much beyond the first few steps is a job for a computer. Suppose, for example, we wanted to determine the Euler approximation to the solution of Eq. (11.19) for chain radioactive decay over the time interval $0 \leq t \leq 50.0$ with a time step of $\Delta t = 0.25$ and $k_A = k_B = 0.1$. We could exploit the capabilities of IDL as follows. First, we set up arrays for $t$, $A$, $B$, and $C$, where 201 elements are needed (time interval divided by time step $= 50.0/0.25 = 200$ steps, but we need an element also for the initial value). Possible statements are

```
IDL> t = fltarr(201)            \text{Create vector for } t.
IDL> A = fltarr(201)            \text{Create vectors for } A(t), B(t), C(t).
IDL> B = fltarr(201)
IDL> C = fltarr(201)
```

Since $t$, $B$, and $C$ start with the value 0.0 (which has already been established in the corresponding vectors), we need set the initial value explicitly only for $A$. We also set values for the two parameters and for the time step. These values are established with the commands

```
IDL> A[0] = 1000.0            \text{Set } A(0) \text{ to initial value.}
IDL> kA = 0.1 & kB = 0.1     \text{Set parameters.}
IDL> dt = 0.25              \text{Set time step.}
```

The solution is then calculated, time instant by time instant, with the loop\footnote{As described at the end of Section 2.8, when we work interactively, the commands in an IDL \texttt{for} loop must be assembled so that IDL sees the \texttt{ent}ire loop as a \texttt{sing}le command. Thus, the ampersands and dollar signs are critical. The same constraint applies when a \texttt{for} loop is used in a command file executed with the \texttt{@} command. When a \texttt{for} loop is used in a procedure or function file, however, the loop is delimited by the \texttt{begin} and \texttt{endfor} commands and no ampersands or dollar signs are necessary.}

...
Table 11.6: Solution to radioactive decay at $t = 6.0$ for indicated time steps. These solutions were obtained via Euler’s method.

<table>
<thead>
<tr>
<th>$t$</th>
<th>$\Delta t$</th>
<th>$A$</th>
<th>$B$</th>
<th>$C$</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.0000</td>
<td>2.0000</td>
<td>512.000</td>
<td>384.000</td>
<td>104.000</td>
</tr>
<tr>
<td>1.0000</td>
<td>531.441</td>
<td>354.294</td>
<td>114.265</td>
<td></td>
</tr>
<tr>
<td>0.5000</td>
<td>540.360</td>
<td>341.280</td>
<td>118.360</td>
<td></td>
</tr>
<tr>
<td>0.2500</td>
<td>544.641</td>
<td>335.164</td>
<td>120.194</td>
<td></td>
</tr>
<tr>
<td>0.1250</td>
<td>546.740</td>
<td>332.197</td>
<td>121.063</td>
<td></td>
</tr>
<tr>
<td>0.0625</td>
<td>547.779</td>
<td>330.735</td>
<td>121.486</td>
<td></td>
</tr>
</tbody>
</table>

IDL> for i = 1,200 do begin $\$
IDL> B[i] = B[i-1] + kA*A[i-1]*dt - kB*B[i-1]*dt & $\$
IDL> C[i] = C[i-1] + kB*B[i-1]*dt & $\$
IDL> t[i] = t[i-1] + dt & $\$
IDL> endfor

which calculates each new $A$, $B$, and $C$ using the stepping equations given in Eqs. (11.81), (11.82), and (11.83) and each new $t$ simply by adding the time step $dt$ to the previous $t$.\textsuperscript{47} The simple statement

IDL> for i = 0,3 do print, [ t[i], A[i], B[i], C[i], [ A[i]+B[i]+C[i] ] ]

| 0.00000 | 1000.00 | 0.00000 | 0.00000 | 1000.00 |
| 0.25000 | 975.000 | 25.0000 | 0.00000 | 1000.00 |
| 0.50000 | 950.625 | 48.7500 | 0.62500 | 1000.00 |
| 0.75000 | 926.859 | 71.2969 | 1.84375 | 1000.00 |

displays the resulting solution and the conserved quantity $A + B + C$ for the first few time steps. Reassuringly, the values here agree with those in Table 11.2. The more interpretable graphical output of Fig. 11.10 is produced by the statements

IDL> plot, t, A, thick=3.0, title='Three-Species Decay Chain', $\$
IDL> xtitle='Time', ytitle='Number of Atoms Present', charsize=1.3
IDL> oplot, t, B, thick=3.0, linestyle=1
IDL> oplot, t, C, thick=3.0, linestyle=2

To assess the accuracy achieved by Euler’s method more completely, we repeat the above calculation for several different time steps, finding the representative values shown in Table 11.6. The convergence of $A(6.0)$, $B(6.0)$, and $C(6.0)$ as $\Delta t$ is reduced by successive factors of 2 is apparent. Since, even at the end, the approximate solutions still seem to be changing in the units digit with each refinement, we would be off base to claim an accuracy much smaller than $\pm 1.0$ in these values. For some purposes, of course, that accuracy may well be adequate. In particular, to the resolution of the graph in Fig. 11.10, for example, a variation of $\pm 1.0$ in a particular vertical coordinate would hardly show. At the same time, we must remember that these differences may cumulate and may well be larger at $t = 50.0$, for example, than they are at $t = 6.0$. To assess that possibility, we

\textsuperscript{47}For simplicity, we have calculated each new time by adding the time step to the previous time. If we were more concerned about minimizing round-off error, we would be better off replacing the last statement in the loop with the statement $t[i] = i \times dt$, though doing so would increase the execution time. (Multiplication takes more time than addition.)
Figure 11.10: Solution of chain radioactive decay via Euler’s method with $\Delta t = 0.25$. The solid, dotted, and dashed lines show $A(t)$, $B(t)$, and $C(t)$, respectively.

Table 11.7: Solution to radioactive decay at $t = 6.0$ for indicated time steps. These solutions were obtained via the improved Euler method.

<table>
<thead>
<tr>
<th>$t$</th>
<th>$\Delta t$</th>
<th>$A$</th>
<th>$B$</th>
<th>$C$</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.0000</td>
<td>2.0000</td>
<td>551.368</td>
<td>322.752</td>
<td>125.880</td>
</tr>
<tr>
<td>1.0000</td>
<td>1.0000</td>
<td>549.404</td>
<td>327.821</td>
<td>122.776</td>
</tr>
<tr>
<td>0.5000</td>
<td>0.5000</td>
<td>548.954</td>
<td>328.940</td>
<td>122.106</td>
</tr>
<tr>
<td>0.2500</td>
<td>0.2500</td>
<td>548.846</td>
<td>329.202</td>
<td>121.951</td>
</tr>
<tr>
<td>0.1250</td>
<td>0.1250</td>
<td>548.820</td>
<td>329.266</td>
<td>121.914</td>
</tr>
<tr>
<td>0.0625</td>
<td>0.0625</td>
<td>548.814</td>
<td>329.282</td>
<td>121.904</td>
</tr>
</tbody>
</table>

generate the entire solution via Euler’s method for three time steps ($\Delta t = 2.0$, 0.5, and 0.0625), plotting those solutions in Fig. 11.11. The comparison of the resulting graphs is described in the caption to the figure.

A similar demonstration using the improved Euler method is left to an exercise. In particular, the values presented in Table 11.7 reflect the result of that exercise when the procedures leading to Table 11.6 are repeated with the improved Euler method. Clearly, the results in Table 11.7 are converging more quickly on stable values to more decimal places than we observed in Table 11.6. Remember that, with Euler’s method, halving $\Delta t$ reduces the global truncation error by a factor of two while, with the improved Euler method, halving $\Delta t$ reduces the global truncation error by a factor of four. Said another way, we would have to reduce $\Delta t$ by a factor of ten to gain one decimal digit in a solution by Euler’s method; we would have to reduce $\Delta t$ by only a factor of $\sqrt{10} = 3.2$ to gain a decimal digit in a solution by the improved Euler method. The values in these two tables reflect this difference.
Figure 11.11: Solution of chain radioactive decay via Euler’s method. The solid, dotted, and dashed lines show the solution for $\Delta t = 0.0625$, 0.5, and 2.0, respectively. While reducing $\Delta t$ from 2.0 to 0.5 clearly makes a difference on the scale of the graph, the further reduction from 0.5 to 0.0625 is hardly noticeable over the entire range of the independent variable.

![Three-Species Decay Chain](image)

Table 11.8: A generic function to define ODEs for IDL.

```idl
FUNCTION FunctionName, IndVar, DepVar
    ;+    Explanatory comments.
    ;-    common params, NameOfParam1, NameOfParam2, ...
    ;
    Intermediate statements.
    ;
    RETURN, [DerivOfFirstVar, DerivOfSecondVar, ...]
END
```

11.7.2 Defining ODEs for IDL

Before using any of the routines IDL supplies for solving a system of first-order differential equations numerically, we must create a function pro-file\(^48\) whose execution by IDL returns the derivatives of the dependent variables as defined by the differential equations we seek to solve. Fortunately, all routines use the same structure for this pro-file, which—in broad outline—will have the general form shown in Table 11.8. In this structure,

\(^48\)See Section 2.8.
• The function name must be the same as the name of the file in which it is stored. For example, if we choose to call the function for the radioactive decay chain decay, then the file storing the function must be named decay.pro.

• The variables must be given in the order independent variable, dependent variable.

• If the differential equations of interest have no internal parameters, then the common area named params should be omitted from the file defining the function.49

Explicit pro-files for several examples will be presented in the subsequent subsections.

11.7.3 The Procedures rk4, lsode, ludiffeq.23, and ludiffeq.45

IDL contains two built-in procedures for solving ODEs. The procedure50 rk4 is adaptive and uses a fourth-order Runge-Kutta method to step the solution forward by a specified amount, holding the solution to an absolute (truncation) error per step no larger in absolute value than $10^{-6}$. A statement invoking rk4 has the general form

\[
\text{soln} = \text{rk4}( \text{VectorOfDepVars}, \text{VectorOfDerivs}, \text{IndVar}, \text{TimeStep}, '\text{FunctionName}')
\]

where the first two arguments are vectors containing the solution and the derivatives of the solution at the value of the independent variable provided in the third argument, the fourth argument gives the increment by which the independent variable is to be advanced, and the final argument—placed in (single or double) quotation marks—names the function pro-file defining the differential equations. An optional keyword /double will request rk4 to carry out its calculations in double precision.

The second built-in procedure for solving ODEs is lsode, an adaptive procedure that, as with rk4, advances the solution by one time step of specified size.51 A statement invoking lsode has the general form

\[
\text{soln} = \text{lsode}( \text{VectorOfDepVars}, \text{IndVar}, \text{TimeStep}, '\text{FunctionName}', \text{Status}, \text{atol} = \text{AbsTol}, \text{rtol} = \text{RelTol})
\]

where the first argument is a vector containing the solution at the value of the independent variable provided in the second argument, the third argument gives the increment by which the independent variable is to be advanced, and the fourth argument specifies the function pro-file defining the differential equations. The argument Status is an integer, which must be set to the value 1 on the first call to lsode, though the procedure is written to assume that value by default (so the argument can be regarded as optional). Thereafter, this quantity is manipulated by lsode itself and provides a means by which (a) one invocation of the routine can communicate its status to the next invocation and (b) errors encountered in the calculation can be communicated to the user.52 Finally, the two keywords give the user control over the tolerance to which each step in the calculation will be held.53 Specifically, atol (default $10^{-7}$) specifies the absolute value of the maximum difference between successive estimates of the solution at the end of the step,54 and rtol (default $10^{-7}$) specifies

---

49 Use of named common in IDL is discussed more fully in Sections 2.8 and 2.17.


51 The IDL procedure lsode is based on the public-domain FORTRAN program LSODE written by Alan C. Hindmarsh of the Computing and Mathematics Research Division of the Lawrence Livermore National Laboratory. See Appendix Z for full contact information.

52 The interpretation of the possible values of this argument is fully explained in the IDL manuals.

53 Note that these tolerances specify error per step; the global error may well be larger than the error per step. Beware!

54Actually, atol can be a vector if a different absolute tolerance is to be specified for each dependent variable.
the absolute value of the maximum relative difference between successive estimates. Whatever the values, the error per step in the \(i\)-th dependent variable will be kept smaller than \(rtol \cdot \text{abs}(y[i]) + atol\). Either \(atol\) or \(rtol\) can be set to zero if assessment of the error per step is to be done purely by one means or the other.

Since \texttt{rk4} and \texttt{lsode} advance the solution by one step of the specified size, generation of an entire solution from the initial conditions requires us to embed calls to these procedures in an explicitly constructed loop. More specific construction of sample loops will be illustrated in the examples to come.

In 1991, before the built-in routines \texttt{rk4} and \texttt{lsode} became part of IDL, the author, with help from the vendors of IDL, wrote two adaptive ODE routines to solve systems of first-order differential equations for use at Lawrence: \texttt{ludiffeq.23}, which uses a combination of second and third order Runge-Kutta algorithms, and \texttt{ludiffeq.45}, which implements a combination of fourth and fifth order Runge-Kutta algorithms. The algorithms used in these routines are patterned after the algorithms then used in the MATLAB routines \texttt{ode23} and \texttt{ode45}, respectively, and are used in and distributed with this book both by permission of The MathWorks, Inc., which owns the copyright on the original algorithms, and by permission of Research Systems, Inc., whose employees assisted in the coding of the algorithms in IDL’s language. Both \texttt{ludiffeq.23} and \texttt{ludiffeq.45} impose a limit on the minimum size of the time step. If the solution does not converge to the specified tolerance before the minimum time step is reached, the program will print the message, “Beware of Singularity”, though the keyword \texttt{maxsteps} described later gives the user some control over the appearance of this error message.

The two routines \texttt{ludiffeq.23} and \texttt{ludiffeq.45} are implemented as procedures. Each is invoked with the same arguments. Specifically, the form of a call to \texttt{ludiffeq.23} is

\[
\texttt{ludiffeq.23, 'FunctionName', IndVar, DepVars, t0 = StartTime, tf = EndTime, $
\text{"init = InitConds, tol = RelativeErrorTolerance, maxsteps = MaxStepsInSol}$$}
\]

In this statement,

- The statement must begin with the command name (\texttt{ludiffeq.23}), the function name in quotation marks (either single or double), the name of a vector to store the independent variable, and the name of an array to store the dependent variables \textit{in this order}.
- Each of the remaining items is identified by a keyword. Thus, their order is irrelevant.
- Failure to specify a value for the tolerance results in IDL selecting default values of \(10^{-3}\) for \texttt{ludiffeq.23} and \(10^{-6}\) for \texttt{ludiffeq.45}.
- When the keyword \texttt{maxsteps} is given a value, the procedure will calculate the quantity \((tf - t0)/\texttt{maxsteps}\) and interpret the result as a lower limit on the step size to which the built-in adaptive refinement will go as the procedure seeks to achieve the specified (or default) tolerance.\[^{58}\] If this minimum stepsize is reached but the internal tests indicate that the desired tolerance has not been achieved, the procedure will print the message “Beware of Singularity” and terminate execution. Note two features: (1) Failure to specify a value for \texttt{maxsteps} results in IDL selecting the default value 20000. (2) The specified value of \texttt{maxsteps} should not be

\[^{55}\] The MathWorks, Inc., is the vendor of MATLAB. See Appendix Z for full contact information.

\[^{56}\] IDL became commercially available in 1977, at which time the vendor was RSI (Research Systems, Inc.). In 2004, RSI became a wholly owned subsidiary of ITT Industries and was named ITT Visual Information Solutions. In 2011, that company changed its name to Exelis Visual Information Solutions. In 2015, Exelis was purchased by the Harris Corporation and became Harris Geospatial. See Appendix Z for full contact information.

\[^{57}\] The additional (hidden) keyword \texttt{Params}, which is supplied for backward compatibility to an earlier version, is now obsolete and is not here described. The function of this keyword is more easily accomplished through the use of common storage, as described in Section 11.7.2.

\[^{58}\] The largest allowed step size is taken to be \((tf-t0)/5\), but no error results if the procedure leads to a larger step size. The larger step size is simply replaced by the maximum allowed size.
FUNCTION decay, t, n

; The function DECAY defines the rate equations for a three
; species radioactive decay sequence such as
;
; n[0] <--kA--> n[1] <--kB--> n[2]
;
; The parameters kA and kB must be passed to the routine in
; that order in a common area named params.
;
common params, kA, kB
RETURN, [ -kA*n[0], kA*n[0]-kB*n[1], kB*n[1] ]
END

less than 100, since the program begins by supposing a trial step size of \((tf - t0)/100\), and a
trial step size smaller than the smallest allowable step size will result in immediate termination
of execution.

With this information as input, the user invokes \texttt{ludiffeq.23} or \texttt{ludiffeq.45} to generate the
solution and store it in the specified pair of arrays in the IDL workspace. One of these arrays is
a vector containing the values of the independent variable, usually time (e.g., the first column in
Table 11.2). The second array is a two-dimensional matrix that contains values of the dependent
variables (e.g., the second, third, and fourth columns of Table 11.2).

### 11.7.4 Chain Radioactive Decay

As a first example of the use of an IDL routine, consider the equations describing chain radioactive
decay, Eq. (11.19). Before we can construct the proper pro-file to communicate the differential
equations to IDL’s routine, we must establish a correspondence between the elements of a row
vector—name it \(n\)—and the dependent variables. Since we have three dependent variables (the
number of nuclei of species \(A\), \(B\), and \(C\)), \(n\) will have three elements, which IDL refers to as \(n[0]\),
\(n[1]\), and \(n[2]\). If we assign the number of nuclei of species \(A\) to the first element, the number
of nuclei of species \(B\) to the second element, and the number of nuclei of species \(C\) to the third
element, we have the associations \(A \rightarrow n[0]\), \(B \rightarrow n[1]\), and \(C \rightarrow n[2]\). Rewriting the differential
equations Eq. (11.19) in terms of the elements of \(n\), we have the system of equations

\[
\begin{align*}
\frac{dn[0]}{dt} &= -k_A n[0] \\
\frac{dn[1]}{dt} &= k_A n[0] - k_B n[1] \\
\frac{dn[2]}{dt} &= k_B n[1]
\end{align*}
\] (11.103)

Now we are ready to construct the function pro-file describing this system of differential
equations. Respecting the general structure described in Section 11.7.2, we would compose the definition
in Table 11.9. Here, the first line tells IDL that the function is named \texttt{decay} and declares \(t\)—a
scalar—to be the independent variable and \(n\)—a vector—to be the dependent variables. The
common area named \texttt{params} provides the means to communicate the two parameters \(k_A\) and \(k_B\) from
IDL’s main level. The next to last line, which begins with the mandatory word \texttt{RETURN}, specifies
a vector whose elements are the derivatives of the dependent variables as determined by the right
hand sides of Eqs. (11.103). Finally, the last line is a mandatory statement ending the function.
11.7. SOLVING ODES NUMERICALLY WITH IDL

With the function of the previous paragraph stored in the file `decay.pro` in the default directory, we would produce a solution for the differential equations it defines with the statements\(^{59,60}\)

IDL> common params, kA, kB  
Set parameters at command level.

IDL> kA = 0.1 & kB = 0.1  
Set initial conditions.

IDL> ic = [1000.0, 0.0, 0.0 ]  
Set initial conditions.

IDL> ludiffeq_23,'decay', t, n, $  
Solve equations over \(0.0 \leq t \leq 50.0\) placing times in \(t\) and solution in \(n\) and accepting the default values for \(tol\) and \(maxsteps\).

Then we would plot the three dependent variables on the same axes with the statements

IDL> plot, t, n[0,*], title='Three-Species Chain Decay', $  
Set parameters at command level.

IDL> xtitle='Time', ytitle='Number of Atoms Present', $  
Set initial conditions.

IDL> thick=3.0, charsize=1.3  
Solve equations over \(0.0 \leq t \leq 50.0\) placing times in \(t\) and solution in \(n\) and accepting the default values for \(tol\) and \(maxsteps\).

IDL> oplot, t, n[1,*], linestyle=1, thick=3.0  
Set initial conditions.

IDL> oplot, t, n[2,*], linestyle=2, thick=3.0  
Solve equations over \(0.0 \leq t \leq 50.0\) placing times in \(t\) and solution in \(n\) and accepting the default values for \(tol\) and \(maxsteps\).

The resulting graph—Fig. 11.12—shows the time variation of the number of atoms of each species.

Take a few minutes to explore the effects of different decay constants by trying several different values of \(k_A\) and \(k_B\). Take a few minutes also to compare the graph in Fig. 11.10 with the graph in Fig. 11.12. The first of these figures was produced with Euler’s method with a time step of 0.25 and required 201 points in the solution; the second was produced with the second/third order Runge-Kutta method of `ludiffeq_23`, with the (default) tolerance of 0.001, and was accomplished with 26 points, as determined by executing the IDL command

IDL> print, size(t)  
26  
26

after the commands most recently described have been executed. To the resolution of the graphs, the two solutions appear to be pretty much the same. Evidently, the choice of the time step in our application of Euler’s method was adequate to yield a solution accurate to the resolution of the graph, which is the most restrictive statement the evidence we have quoted supports. We might alternatively have inferred the likely appropriateness of the time step \(\Delta t = 0.25\) in Euler’s method by noting from the graphs that the solution varies appreciably over a time period on the order of a few time units. For example, the initial value of \(A\) drops by 20% from 1000 to 800 in about 2 or 3 time units. Since the time step \(\Delta t\) is small compared to that characteristic time, we would be justified in anticipating a reasonably accurate solution.

Let us also generate a solution to this same problem using `rk4`. The function pro-file `decay.pro` will serve in this capacity as well, but we must decide \(a priori\) the instants of time at which we seek a solution and build `rk4` into a loop that is executed for each of those times. Suppose we seek the solution over the time interval \(0 \leq t \leq 50\) again, dividing that interval into 200 steps of size 0.25. We would invoke the following statements:

\(^{59}\)In the illustrated use of IDL, we did not have to compile the pro-file defining the system of differential equations. If, however, our first attempt to invoke any ODE solver generates compilation errors arising from problems with `decay.pro`, we would need to correct those errors in the source file. Then, short of exiting from IDL and reentering, we would need to recompile `decay.pro` explicitly. As described more fully in Section 2.8, the IDL statement to accomplish this recompilation is `.compile decay`. Otherwise, a new invocation of the ODE solver will reuse the already-compiled, erroneous version of the pro-file.

\(^{60}\)Note that we can have more parameters in the `common` statement at IDL’s command level than are actually used in the routines called. Here we have set just the two needed by `decay.pro`. If we anticipated using the common area `params` in other contexts in the same IDL session, we might find it convenient to include some dummy parameters at the creation of that common area. The only way to remove or redefine an already defined common area is to reset the IDL session (see Section 2.17) or exit from and reenter IDL, neither of which may be desirable in some contexts. No harm is done when a common area contains unused parameters, and the “extra” ones need not be mentioned in routines that don’t need them.
Figure 11.12: Solution of chain radioactive decay via `ludiffeq23` with a tolerance of 0.001. The solid, dotted and dashed lines show \(A(t)\), \(B(t)\), and \(C(t)\), respectively.

IDL> t = findgen(201)/4.0
IDL> n = fltarr( 3, 201 )
IDL> n[0,0] = 1000.0
IDL> dt = 0.25
IDL> common params, kA, kB
IDL> kA = 0.1 & kB = 0.1
IDL> for i = 1, 200 do begin $
IDL> \text{der} = \text{decay}(t[i-1], n[*,i-1]) &$
IDL> n[*,i] = rk4( n[*,i-1], \text{der}, t[i-1], dt, 'decay' ) &$
IDL> endfor

Here, the first statement in the loop calculates the derivatives at the current point for use in the second statement, which invokes `rk4` to step the solution from time \(t_{i-1}\) to time \(t_i\). A graph of the resulting solution would be produced by statements identical to those that produced Fig. 11.12, and the resulting graph is essentially identical to Fig. 11.12.

Finally, let us use `lsode` to solve this problem. In essence, we execute the same statements as in the previous paragraph, replacing the loop with the statements

IDL> for i = 1, 200 do begin $
IDL> \text{tmp} = n[*,i-1] &$
IDL> n[*,i] = lsode( \text{tmp}, t[i-1], dt, 'decay' ) &$
IDL> endfor

Here, we do not need an anticipatory evaluation of the derivatives (as was required by `rk4`), but we do need to extract the current values of the dependent variables from the growing array (variable `tmp`) because `lsode` will not accept the entity \(n[*,i-1]\) as its first argument. Plotting statements identical to those that produced Fig. 11.12 will again produce a graph essentially identical to Fig. 11.12.
Table 11.10: The IDL function *damposc.pro*.

```idl
FUNCTION damposc, t, y
;+
; The function DAMPOSC describes the nondimensionalized equations of
; motion for a damped, one dimensional harmonic oscillator. bb is a
; dimensionless parameter and has a value of bb=b/sqrt(m*k), where b
; is the damping, m is the mass and k is the spring constant. This one
; parameter must be passed to the routine in a common area named params.
;-
; common params, bb
RETURN, [ y[1], -y[0] - bb*y[1] ]
END
```

11.7.5 **Damped Harmonic Oscillator**

Consider next the damped harmonic oscillator described in dimensionless form as a pair of first-order
equations by Eq. (11.30), though we shall for simplicity remove the driving force by setting \( f(t) \) to zero. In writing the function pro-file for this system, we first realize that we have two first-order
differential equations. Thus, the array, which we name \( y \), will have two columns and we make the
assignments \( x \mapsto y[0] \) and \( v \mapsto y[1] \). In these terms, the differential equations become

\[
d\frac{dt}{dt} y[0] = y[1] ; \quad d\frac{dt}{dt} y[1] = -y[0] - \beta y[1] \tag{11.104}
\]

Then, using the symbol \( \beta \) for \( \beta \), we create the pro-file recorded in Table 11.10 defining the differential
equations. After storing this file in the default directory with the name *damposc.pro*, we explore
the system for two different damping constants with the statements

```idl
IDL> ic=[1.0,0.0] ; Set initial conditions.
IDL> common params, bb ; Define common for parameters.
IDL> bb=0.0 ; Set damping to zero.
IDL> ludiffeq_23, 'damposc', t1, x1,$
IDL> init=ic, t0=0.0, tf=25.0, $ ; Solve equations, putting times in t1 and
IDL> tol=0.0001 ; solution in x1.
IDL> bb=0.25 ; Set small damping.
IDL> ludiffeq_23, 'damposc', t2, x2,$
IDL> init=ic, t0=0.0, tf=25.0,$ ; Solve equations with same ic's as before but
IDL> tol=0.0001 ; putting times in t2 and solution in x2.
```

Finally, we produce graphs showing the position and velocity as functions of time and also the
trajectory in the phase plane \( (v \text{ versus } x) \) for each value of the damping constant with the statements

```idl
IDL> !p.multi = [ 0, 2, 2 ]
IDL> plot, t1, x1[0,*], thick=2.0, title='Undamped Harmonic Oscillator', $
IDL> subtitle='X and V as Functions of T', xtitle='Time'
IDL> oplot, t1, x1[1,*], linestyle=2, thick=2.0
IDL> plot, t2, x2[0,*], thick = 2.0, title='Damped Harmonic Oscillator', $
IDL> subtitle='X and V as Functions of T', xtitle='Time'
```
Figure 11.13: Damped and undamped harmonic oscillators. Notice the difference between the phase plane orbits.

Alternatively, we could, of course, approach this problem using \texttt{rk4} or \texttt{lsode}. Deciding to seek a solution over the interval $0 \leq t \leq 25$ and dividing the interval into 100 segments, we set up the variables with the statements

\begin{verbatim}
IDL> t = findgen(101)/4.0
IDL> soln = fltarr(2,101)
IDL> soln[0,0] = 1.0
IDL> dt = 0.25
IDL> common params, bb
IDL> bb = 0.0
\end{verbatim}

Then, we pursue the solution either with the loop
11.7. SOLVING ODES NUMERICALLY WITH IDL

IDL> for i = 1, 100 do begin $
IDL> \text{der} = \text{damposc}( t[i-1], \text{soln}(*,i-1) ) &$
IDL> \text{soln}[*,i] = \text{rk4}( \text{soln}[*,i-1], \text{der}, t[i-1], \text{dt}, '\text{damposc}' ) &$
IDL> \text{endfor}

or with the loop

IDL> for i = 1, 100 do begin $
IDL> \text{tmp} = \text{soln}[*,i-1] &$
IDL> \text{soln}[*,i] = \text{lsode}( \text{tmp}, t[i-1], \text{dt}, '\text{damposc}' ) &$
IDL> \text{endfor}

Repeating these steps with $b \text{b}$ set to 0.25 solves the problem for a small damping. Graphs of all of these results would look just like the graphs in Fig. 11.13. Evidence of the accuracy of these solutions can be found in the preservation of the amplitude of the motion for the undamped case, in the fact that the phase-plane diagram for the undamped case traces the same path four times over without evident departure, and in the specified (default) tolerance with which \text{ludiffeq}._45\text{pursues} its solution. Further evidence could be garnered by comparing these solutions with those we might obtain with smaller or larger time steps.

11.7.6 Planetary Orbits

Expressed in dimensionless form in Cartesian coordinates, the equations of motion for a planet of mass $m$ orbiting a sun of mass $M$ were presented in Eq. (11.46). We must, however, recast the pair of second-order equations as a quartet of first-order equations. Thus, we view the equations as the system

\[
\frac{dx}{dt} = \nu_x; \quad \frac{d\nu_x}{dt} = -\frac{x}{(x^2 + y^2)^{b}}; \quad \frac{dy}{dt} = \nu_y; \quad \frac{d\nu_y}{dt} = -\frac{y}{(x^2 + y^2)^{b}} \tag{11.105}
\]

Here, for the sake of generality, we have introduced the parameter $b$, which will allow us to explore forces other than the inverse square force. For the inverse square force, we simply set the parameter to the value 1.5.

To create the necessary pro-file, we set $x$, $\nu_x$, $y$, and $\nu_y$ into correspondence with the elements $x[0]$, $x[1]$, $x[2]$, and $x[3]$ of the four-element vector $x$. Then, one possibility for the pro-file \text{planet.pro} defining the above equations is shown in Table 11.11. Finally, to calibrate our sense of appropriate initial conditions, we remember from Eq. (11.50) that, in our dimensionless units, a circular orbit is achieved when the speed of the planet is the reciprocal of the square root of the radius of the planet’s orbit. For example, a circular orbit of radius 4.0 units requires a speed of $1/\sqrt{4.0} = 0.5$ units.

Now we have all the elements in place to invoke IDL and pursue solutions to the planetary problem. To achieve adequate accuracy, we here need the better algorithm and higher tolerance of \text{ludiffeq}._45. We solve the differential equations for an inverse square law force ($b = 1.5$) twice, first for a circular orbit and then for an elliptical orbit, with the statements

IDL> \text{common params, b}
IDL> b=1.5
IDL> ic1=[ 4.0, 0.0, 0.0, 0.5 ]
IDL> \text{ludiffeq}._45, 'planet', t1, x1, init=ic1, t0=0.0, tf=60.0
IDL> ic2=[ 4.0, 0.0, 0.0, 0.3 ]
IDL> \text{ludiffeq}._45, 'planet', t2, x2, init=ic2, t0=0.0, tf=60.0
FUNCTION planet, t, x
;+  ; The function PLANET describes the dimensionless equations of
;+  ; motion for a planet of mass m orbiting a sun of mass M. The
;+  ; parameter b allows the user to explore forces that are not
;+  ; inverse-square (For an inverse-square law b=1.5.) Entries
;+  ; in the array of dependent variables are X-position, X-velocity,
;+  ; Y-position, Y-velocity. The one parameter must be passed to
;+  ; the routine in a common area named params.
;+-  common params, b
  temp=( x[0]^2 + x[2]^2 )^b
  return, [ x[1], -x[0]/temp, x[3], -x[2]/temp ]
end

Figure 11.14 shows the variety of ways the results of this calculation can be displayed. That the first
case indeed yields a circular orbit provides some evidence supporting the conclusion that we have
generated the solution with adequate accuracy.61

With this example, assessing accuracy by checking the conservation of energy [given by the
first member of Eq. (11.53)] and angular momentum [given by the second member of Eq. (11.53)] is
worthwhile. The statements

IDL> E1 = 0.5*(x1[1,*]^2+x1[3,*]^2) - 1.0/sqrt(x1[0,*]^2+x1[2,*]^2)
IDL> L1 = x1[0,*]*x1[3,*] - x1[2,*]*x1[1,*]
IDL> E2 = 0.5*(x2[1,*]^2+x2[3,*]^2) - 1.0/sqrt(x2[0,*]^2+x2[2,*]^2)
IDL> L2 = x2[0,*]*x2[3,*] - x2[2,*]*x2[1,*]

will calculate these quantities. Then, statements like

IDL> print, [ E1, L1 ]

will display the resulting values. We find—see your own output—that $E_1 = -0.125000$ at the
beginning and climbs to $-0.125001$ by the end of the interval while $L_1$ starts at $2.00000$ and falls
to $1.99999$; $E_2$ varies between $-0.205001$ and $-0.204994$ while $L_2$ remains rock solid at $1.20000$
throughout the interval of solution. These results give us no concern at all about the adequacy of
the solutions, at least insofar as their respect for known conserved quantities is concerned.

Open orbits, in which a satellite with positive energy moves in the gravitational field of a sun,
are also possible. We utilize IDL’s for loop to produce a variety of open orbits for a planet started
in different initial positions by executing the statements

IDL> b=1.5
IDL> plot, [ -4, 4 ], [ -4, 4 ], /nodata, $  
IDL> title='Open Orbits', xtitle='X-Position', ytitle='Y-Position', $  
IDL> charsize=1.3
IDL> for i=-4,4 do begin ic=[-4.0, 1.0, float(i), 0.0] & $  
IDL>   ludiffeq_45, 'planet', t, x, t0=0.0, tf=30.0, init=ic & $  

61 The statements creating the graphs have been omitted but, except for the wrinkle of making circles appear as
  circles (see Section 2.16.5), could be constructed with little difficulty.
Figure 11.14: Circular and elliptical orbits for a planet orbiting a sun. The orbits are shown in the $xy$ plane and in two different phase planes.
11.7.7 Standing Waves in a String

To address a boundary value problem, we must be clever, since the methods available for solving ODEs all suppose that we are dealing with an initial value problem. One strategy (sometimes called a shooting method) involves accepting the boundary value at one end of the interval, guessing a derivative at that same end, assuming values for any parameters in the equation, solving the resulting initial value problem, assessing the extent to which the solution so generated respects the boundary value at the other end of the interval, and repeating the process while tampering with the derivative or, more commonly, with the parameters until satisfactory agreement with both boundary values has been obtained. For standing waves in a string [Eqs. (11.56) and (11.57)] or the quantum particle in an infinitely deep square well [Eq. (11.59)], the expression of the relevant equations as a system of first-order equations is

\[
\frac{df}{dx} = g \quad \frac{dg}{dx} = -k^2 f \quad f(0) = 0 \quad f(\ell) = 0 \tag{11.106}
\]

Thus, setting up the correspondences \( f \mapsto f[0] \) and \( g \mapsto f[1] \), we might define this system for IDL with the pro-file shown in Table 11.12 which we store in the default directory with the name stdwaves.pro.

Adopting the strategy described in the previous paragraph, we would then accept the requirement that \( f(0) = 0 \), assume a value—say 1.0—for \( \frac{df}{dx}(0)/dx = g(0) \) and a value for \( k \)—say 1.0, generate the solution over \( 0 \leq x \leq 1 \), and see whether it returns to the value \( f(1) = 0 \) (where—because we need numbers to effect a numerical solution—we have supposed \( \ell = 1 \), equivalent to recasting...
11.7. SOLVING ODES NUMERICALLY WITH IDL

Table 11.12: The IDL function stdwaves.pro.

```idl
FUNCTION stdwaves, x, f

; The function STDWAVES defines the basic equations for describing
; standing waves in a string. The one parameter k in the equations
; must be passed to the routine in a common area named params.

; common params, k
RETURN, [ f[1], -k^2*f[0] ]
END
```

Figure 11.16: Shooting method to solve for standing waves in a string. The solid, dotted, dashed, and dot-dashed curves correspond to \( k = 1.0, 2.0, 3.0, \) and \( 4.0 \), respectively.

the equation in dimensionless form with \( \ell \) chosen as the unit of length). Choosing `ludiffeq_23`, we would invoke the statements

```idl
IDL> common params, k
IDL> k = 1.0
IDL> ic = [0.0, 1.0]
IDL> ludiffeq_23, 'stdwaves', t, $f, t0=0.0, tf=1.0, init=ic
IDL> plot, t, f[0,*]
```

Set parameter.

Set initial conditions.

Invoke solver.

Plot solution.

The result is the solid line in Fig. 11.16. Repeating this calculation while overriding the default tolerance (0.001) by stipulating a tolerance of 0.0001 for `ludiffeq_23` (do it!), we find no significant difference (to the resolution of the graph), nor do we find significant differences when the two solutions are repeated with \( k = 20.0 \) (again, do it!), so we accept the default tolerance for subsequent calculations.
Clearly, however, the solution we have obtained (and in which we now have reasonable confidence) fails utterly to return to the value zero at the upper end of the interval. Either the assumed derivative \[ g(0) = 1 \] or the assumed value \( k = 1.0 \) is not appropriate. Because the equations are linear, however, changing \( g(0) \) will merely influence the scale of the resulting solution; a different value of the derivative at the beginning will never cause the non-zero value at the end to become zero. We quickly conclude that efforts to bring \( f(1) \) to the value zero stand a chance of succeeding only if we tamper with \( k \). Thus, we explore other values of \( k \) with the statements

```idl
IDL> !y.range=[-1.0,1.0]
IDL> k = 1.0 & ludiffeq_23, 'stdwaves', t, f, t0=0.0, tf=1.0, init=ic
IDL> plot, t, f[0,*], thick=3.0, ticklen=1.0
IDL> k = 2.0 & ludiffeq_23, 'stdwaves', t, f, t0=0.0, tf=1.0, init=ic
IDL> oplot, t, f[0,*], thick=3.0, linestyle=1
IDL> k = 3.0 & ludiffeq_23, 'stdwaves', t, f, t0=0.0, tf=1.0, init=ic
IDL> oplot, t, f[0,*], thick=3.0, linestyle=2
IDL> k = 4.0 & ludiffeq_23, 'stdwaves', t, f, t0=0.0, tf=1.0, init=ic
IDL> oplot, t, f[0,*], thick=3.0, linestyle=3
```

All four tentative solutions are displayed in Fig. 11.16. Evidently, somewhere between \( k = 3.0 \) and \( k = 4.0 \), the solution will assume the proper value at \( x = 1 \), and we are now in a position via manual trial and error to seek more refined estimates of that specific value of \( k \).

Having now concluded that we should be adjusting \( k \) as we seek acceptable solutions and having recognized that the value of the solution at \( x = 1 \) is critical, we imagine that a graph showing \( f(1) \) as a function of \( k \) might help us in deciding where we should look in an effort to find additional solutions. Suppose we were to seek solutions for values of \( k \) ranging from 0 to 20 (say) in steps of 0.2. We might then solve the problem for each \( k \) but save only the value \( f(1) \) for each solution. To generate that information, we would have to create a loop that migrated through the values of \( k \), calculated the solution at each value, and saved the value \( f(1) \). We might use the statements

```idl
IDL> k1 = findgen(101)/5.0
IDL> soln = fltarr(101)
IDL> for i = 0, 100 do begin $
IDL> k = k1[i] &$
IDL> ludiffeq_23, 'stdwaves', t, f, t0=0.0, tf=1.0, init=ic &$
IDL> q = size(t) & soln[i] = f[ 0, q[1]-1 ] &$
IDL> endfor
IDL> plot, k1, soln, ticklen=1.0, thick=3.0
```

Here, we (1) establish values for \( k \), (2) create an array for the values to be saved, (3) execute a loop that (a) copies the next \( k \) into the common area, (b) invokes \texttt{ludiffeq.23} to solve the equations, (c) uses \texttt{size} to determine the number of entries in the arrays containing the solution, and (d) stores the final value (which will be in row \( q[i]-1 \)), and (4) plot the resulting values.

The resulting graph is shown in Fig. 11.17. Each of the zeroes of the function shown in this graph corresponds to a value of \( k \) at which an acceptable solution to the boundary value problem can be found. Further, an approximation to each solution can be read from this graph and taken as input for a more sophisticated search procedure (which, however, we shall not develop here; see Chapter 14). Even more, we might infer from the oscillatory nature of this graph that the problem actually has a very large number of distinct solutions, only the first six of which are identifiable in the range \( 0 \leq k \leq 20 \).
11.7. The Quantum Harmonic Oscillator

The quantum harmonic oscillator described by Eqs. (11.60) and (11.61) provides a second example of a boundary value problem. This one, however, involves an infinite domain. As we suggested earlier, we can generate a more tractable approach by recognizing that the character of the equation compels solutions to have either even or odd parity. Thus, we can focus attention on only the interval \(0 \leq y < \infty\). To set up the problem, we must first regard the second-order equation as a pair of first-order equations in which we associate \(\psi[0]\) with \(\psi\) and \(\psi[1]\) with \(d\psi/dy\). The equations we must solve then become

\[
\frac{d}{dy}\psi[0] = \psi[1]; \quad \frac{d}{dy}\psi[1] = -(2\epsilon - y^2)\psi[0] \tag{11.107}
\]

and a suitable pro-file `qmshm.pro` defining these equations for IDL’s solvers is recorded in Table 11.13. With this file stored in the default directory with the name `qmshm.pro`, we are ready to seek its solutions.

We seek first solutions of even parity, i.e., solutions for which \(\psi(0) \neq 0\) and \(d\psi(0)/dy = 0\). With those values, we can address this boundary value problem as if it were an initial value problem—except that we must reject solutions that do not go to zero as \(y \to \infty\). Only \(\psi(0)\) and \(\epsilon\) are adjustable as we seek to impose the boundary condition at infinity. Since the ODE we are solving is linear, however, we recognize right away that tampering with \(\psi(0)\) will merely affect the scaling of the solution and has no power to convert a solution that doesn’t go to zero at infinity into one that does go to zero. The only parameter we need bother adjusting is \(\epsilon\). Let us, therefore, standardize by setting \(\psi(0) = 1.0\) and explore the dependence of the solution on \(\epsilon\). At the outset, we do not know what to expect. We do, however, believe that, in a dimensionless casting, important quantities are likely to have values on the order of one. Thus, we begin a search for acceptable solutions by setting \(\epsilon = 1.0\) and examining the solution in the interval \(0.0 \leq y \leq 2.0\). The statements

---

**Figure 11.17:** Graph of \(f(1)\) versus \(k\) for standing waves in a string.
CHAPTER 11. SOLVING ORDINARY DIFFERENTIAL EQUATIONS

Table 11.13: The IDL function qmshm.pro.

```idl
FUNCTION qmshm, y, psi
;
; QMSHM defines the dimensionless Schrödinger equation for the quantum
; simple harmonic oscillator, which is
; 2
; d psi 2
; ----- + (2 ee - y ) psi = 0
; 2
; dy
;
; Entries are psi and d psi/ds, and initial conditions will thus be
; [psi0, dpsi0]. The parameter ee represents a dimensionless energy
; and must be passed to the routine in a common area named params.
;
common params, ee
RETURN, [ psi[1], -(2*ee-y^2)*psi[0] ]
END
```

IDL> common params, ee
IDL> ic = [ 1.0, 0.0 ] & ee = 1.0
IDL> ludiffeq_23, 'qmshm', y, psi, t0=0.0, tf=2.0, init=ic
IDL> plot, y, psi[0,*]

generate that solution and produce a graph which—see your own output—starts off sensibly by
decaying from $\psi(0) = 1.0$ towards $\psi(y) = 0.0$ as $y$ increases, but then crosses the value $\psi(y) = 0.0$

at about $y = 1.2$ and gives every indication of heading off towards $-\infty$. With $\epsilon = 1.2$, the divergence

happens more quickly but with $\epsilon = 0.8$, the solution takes longer to diverge. Evidently, lowering

$\epsilon$ from 1.0 moves in the right direction. With this background, we are tempted to examine the

behavior with several values of $\epsilon$ by executing the statements

IDL> e = [ 0.3, 0.4, 0.5, 0.6, 0.7 ]
IDL> q = size(e)
IDL> for i = 0, q[1]-1 do begin $
IDL> ee = e[i] & $
IDL> if i eq 0 then plot, y, psi[0,*], yrange=[-2.0,2.0], thick=3 $
IDL> else oplot, y, psi[0,*], thick=3 & $
IDL> endfor
IDL> oplot, [0.0,4.0], [0.0, 0.0]
IDL> xyouts, 2.0, 1.7, '0.3'
IDL> xyouts, 2.8, 1.2, '0.4'
IDL> xyouts, 3.5, 0.1, '0.5'
IDL> xyouts, 2.8, -1.0, '0.6'
IDL> xyouts, 2.2, -1.6, '0.7'

which step $\epsilon$ through the indicated values, plot the solution at each step, add a horizontal line at $\psi = 0.0$, and label each graph. From the resulting display, which is shown in Fig. 11.18, we conclude

that $\epsilon = 0.5$ yields a solution that goes to zero at infinity and we therefore argue that one allowed

energy corresponds to the value $\epsilon = 0.5$. 

11.7. **SOLVING ODES NUMERICALLY WITH IDL**

Figure 11.18: The solution to the Schrödinger equation for an even state of the quantum harmonic oscillator when $\epsilon$ has the indicated values. This state is the ground state (or state of lowest energy).

If we now allow $\epsilon$ to increase beyond the value 0.7, we find the solution diverges more and more rapidly for a time, but then begins to turn back towards the axis. Identically the same coding as illustrated above, but with the starting line

IDL> e = [ 1.0, 1.5, 2.0, 2.5, 3.0, 3.5 ]

to set the values of $\epsilon$ and the lines

IDL> xyouts, 2.1, -1.2, '1.0'
IDL> xyouts, 1.7, -1.7, '1.5'
IDL> xyouts, 2.7, -1.8, '2.0'
IDL> xyouts, 3.5, -0.3, '2.5'
IDL> xyouts, 2.7, -1.8, '2.0'
IDL> xyouts, 3.0, 1.2, '3.0'
IDL> xyouts, 2.5, 1.7, '3.5'

to place the labels appropriately, produces the graph shown in Fig. 11.19. From this graph, we conclude that $\epsilon = 2.5$ yields another solution that goes to zero at infinity, and we argue that another allowed energy corresponds to the value $\epsilon = 2.5$. We might even be tempted to speculate—correctly as it turns out—that additional allowed energies for **even** states will correspond to the values $\epsilon = 4.5, 6.5, 8.5, \ldots$.

We turn next to the odd states, for which we set $\psi(0) = 0.0$ and $d\psi(0)/dy = 1.0$ before exploring solutions for various values of $\epsilon$. Indeed, the previous paragraphs suggest that we might expect to find the lowest **odd** state corresponding to $\epsilon = 1.5$. To test that expectation, we set the initial conditions and energies with the statements

IDL> ic = [0.0, 1.0]
IDL> e = [ 1.3, 1.4, 1.5, 1.6, 1.7 ]
Figure 11.19: The solution to the Schrödinger equation for an even state of the quantum harmonic oscillator when $\epsilon$ has the indicated values. As we shall confirm in the next figure, this state is actually the second excited state of the harmonic oscillator.

![Graph](image)

Then we execute the statements

IDL> q = size(e)
IDL> for i = 0, q[1]-1 do begin $
IDL> ee = e[i] &$
IDL> ludiffeq_23, 'qmshm', y, psi, t0=0.0, tf=4.0, init=ic &$
IDL> if i eq 0 then plot, y, psi[0,*], yrange=[-2.0,2.0], thick=3$
IDL> else oplot, y, psi[0,*], thick=3 &$
IDL> endfor$
IDL> oplot, [0.0,4.0], [0.0, 0.0]

IDL> xyouts, 2.7, 1.7, '1.3'
IDL> xyouts, 3.5, 1.5, '1.4'
IDL> xyouts, 3.5, 0.1, '1.5'
IDL> xyouts, 3.4, -1.0, '1.6'
IDL> xyouts, 2.8, -1.5, '1.7'

to generate the graph in Fig. 11.20 and the statements

IDL> xyouts, 2.7, 1.7, '1.3'
IDL> xyouts, 3.5, 1.5, '1.4'
IDL> xyouts, 3.5, 0.1, '1.5'
IDL> xyouts, 3.4, -1.0, '1.6'
IDL> xyouts, 2.8, -1.5, '1.7'

to label the individual curves in that graph. That the graph for $\epsilon = 1.5$ behaves properly at infinity confirms our suspicion that the first odd state occurs at that value of the energy. We further suspect that additional odd states will be found at $\epsilon = 3.5, 5.5, 7.5, \ldots$.

Finally, in broad terms, we conclude that the ground state of the quantum harmonic oscillator has even parity; that acceptable solutions occur when $\epsilon = n + 1/2$ with $n = 0, 1, 2, 3, \ldots$; and that odd and even states occur alternately as $n$ increases, i.e., that the parity of the $n$-th state is $(-1)^n$. 

Finally, remembering that the physical energy $E$ and the dimensionless energy $\epsilon$ are related by $E = \epsilon \hbar \omega$, we infer that the energies of the quantum harmonic oscillator are given by

$$E_n = \left( n + \frac{1}{2} \right) \hbar \omega \quad (11.108)$$

though we have direct evidence for this (correct) conclusion only for the lowest three states.

### 11.7.9 Pro-files in the Public Library

The following pro-files are found in the directory `$/HEAD/idl` (and may also at some sites have been placed in the directory `$/IDLHEAD/lib/lu`, which is automatically in the IDL search path) and can be used to explore the systems they describe. Further information about the file `decay.pro`, for example, can be obtained either with the command

```
IDL> doc_library, 'decay', directory='$/HEAD/idl'
```

to IDL or with whatever command to your operating system lists the file `$/HEAD/idl/decay.pro` on the screen.\(^{65}\) The first of these commands instructs IDL to find the file `decay.pro` and display the comments between the line `*+` and the line `*-` at the beginning of the file; the second command will display the entire file on the screen. Many of the functions defined by these files involve parameters that must be supplied in a common area named `params`; examine the help message and/or the file itself to determine the structure of the necessary common area.

---

\(^63\)While it is no longer allowed, older versions of IDL permitted the specification of the path as part of the file name, e.g., `doc_library, '$/HEAD/idl/decay'`.  
\(^64\)The keyword `directory` is available only in a UNIX or LINUX operating system.  
\(^65\)In UNIX, one possible command would be `more $/HEAD/idl/decay.pro`; see the `Local Guide`.  

• **2oscil.pro**, which describes a system of two equal masses between two walls, all connected by three springs. Dimensionless units are used.

• **apollo.pro**, which describes a satellite in the gravitational attraction of two fixed suns of equal mass.

• **dampharm.pro**, which describes the one-dimensional damped, unforced harmonic oscillator using dimensional variables.

• **damposc.pro**, which describes the one-dimensional, damped, unforced harmonic oscillator using dimensionless variables.

• **decay.pro**, which describes three-species radioactive decay.

• **drvnosc.pro**, which describes the one-dimensional, damped, forced harmonic oscillator using dimensional variables.

• **drvp pend.pro**, which describes the forced, damped simple pendulum in dimensionless units.

• **henon.pro**, which describes the Hénon-Heiles oscillator.

• **largamp.pro**, which describes the undamped, unforced large amplitude pendulum in dimensionless units.

• **lorenz.pro**, which describes the Lorenz attractor.

• **onedshm.pro**, which describes the one-dimensional, undamped, unforced harmonic oscillator using dimensional variables.

• **planet.pro**, which describes the motion of a planet of mass $m$ around a sun of mass $M$, using dimensionless units. An additional parameter allows for exploration of non-inverse square forces.

• **qms shm.pro**, which describes the dimensionless quantum harmonic oscillator.

• **rossler.pro**, which describes the Rössler attractor.

• **stdwaves.pro**, which describes the behavior of standing waves in a string.

• **twodshm.pro**, which describes the two-dimensional, undamped, unforced harmonic oscillator using dimensionless variables.

• **vandpol.pro**, which describes the Van der Pol oscillator.

### 11.12 Solving ODEs Numerically with MAPLE

MAPLE’s command `dsolve` can be forced into providing a means to solve ordinary differential equations numerically by specifying an optional argument that selects `numeric` as the desired `type` for the solution. With this keyword, a full statement will have the form

```
dsolve( { ODE1, ODE2, ..., IC1, IC2, ...}, { DVar1(IVar), DVar2(IVar), ...},
        type = numeric )
```
where $ODE$ and $IC$ are the differential equations and initial conditions, $DVar$ represents the dependent variables, and $IVar$ represents the independent variable. In this case, both the differential equations and the initial conditions must be specified; MAPLE will not introduce undetermined integration constants. Further, all parameters in the equations must be given explicit numerical values before $dsolve$ is executed. When invoked, $dsolve$ returns not a solution but a procedure whose invocation will generate the numerical value of the solution (once a specific numerical value has been assigned to the independent variable).

Thus, for example, to address the problem of chain radioactive decay that we have already solved symbolically in Section 11.4.4, we would request MAPLE to generate a suitable procedure by invoking the statements

```maple
> eqA := Diff( A(t), t ) = -kA*A(t):
Enter equations, suppressing display.
> eqB := Diff( B(t), t ) = kA*A(t)-kB*B(t):
> eqC := Diff( C(t), t ) = kB*B(t):
> kA := 0.1: kB := 0.1:
> soln := dsolve( { eqA, eqB, eqC,
A(0)=1000, B(0)=0, C(0)=0 },
{ A(t), B(t), C(t) }, type=numeric );
    Request numeric solution, incorporating
    initial conditions in request.

soln := proc(x_rkf45) . . . end proc
```

Then, after the procedure has been created, parameters can be assigned and the solution at any particular value of the independent variable can be found with statements like

```maple
> soln(0.0);
    Enter equations, suppressing display.
[t = 0, A(t) = 1000., B(t) = 0., C(t) = 0.]
> soln(0.1);
[t = 0.1, A(t) = 990.0498337..., B(t) = 9.9004983..., C(t) = .0496679...]
```

(The ellipses in the last statement stand for several additional digits returned by MAPLE.) Note the form of the solution, which is presented as a list of equations, the first of which conveys the applicable value of the independent variable and the rest of which convey the corresponding values of the several dependent variables.

Should we wish a graph of the solution over some range of the independent variables, we would invoke the command $odeplot$ in the $plots$ package—a command that is able to interpret the structure of the solution returned by the procedure created by $dsolve$. A graph similar to Fig. 11.9 for this decay would, for example, be displayed by the statement

```maple
> plots[odeplot](soln, [ [t, A(t)], [t, B(t)], [t,C(t)] ], 0..50,
color=black, thickness=3 );
```

Additional keywords similar to those available for the command $plot$ can be exploited to add elements to the graph and/or override additional defaults.

The previous two paragraphs illustrate the essential strategy adopted by MAPLE to produce numerical solutions to ordinary differential equations. We here draw attention to a few additional capabilities, leaving an exhaustive enumeration and the details to the MAPLE manuals:

- By default, MAPLE generates a procedure that uses a fourth-fifth order Runge-Kutta-Fehlberg (RKF) method. Other methods can be selected by specifying the keyword $method$, e.g., $method=choice$ (where $choice$ has many options, including $rKF45$, $bvp$, $lsode$, and $taylorseries$) or $method=classical[choice]$ (where $choice$ has many options, including $foreuler$, $rk2$, $rk4$ and $abmoulton$). See the MAPLE manuals under $dsolve/numeric$ and $dsolve/numeric/classical$ for descriptions of these and other options.

---

66Note that the reported name of the procedure used may vary with the version of MAPLE.
• By default for the RKF method, the absolute tolerance per step is set to $10^{-7}$ and the relative tolerance per step is set to $10^{-6}$, but these values can be changed by using the keywords `abserr`, which establishes a limit on the absolute error per step, and `relerr`, which establishes a limit on the relative error per step.

• If we want solutions at a specific set of values of the independent variable, we could exploit the keyword `output`, whose value is set to an array that contains the values of the independent variable at which solutions are desired. The solution is then returned as a matrix with one column and two rows. The element in the first row is a row vector containing the names of the dependent and independent variables and the element in the second row is a matrix containing one row for each value of the independent variable. Each of those rows contains the value of the independent variable and the corresponding values of the dependent variables. Thus, for example, the statements

```maple
> kA := 0.1; kB := 0.1:
> soln1 := dsolve( { eqA, eqB, eqC, A(0)=1000, B(0)=0, C(0)=0 },
                   { A(t), B(t), C(t) }, type=numeric,
                   output=array( [0.0, 5.0, 10.0, 15.0, 20.0] ) );
```

will produce the output

```
soln1 :=
    0.          1000.          0.
   5.0       606.5306579...       303.2653385...    90.20400350...
  10.0      367.8794362...     367.8794642...     264.2410994...
 15.0     223.1301494...     334.6952874...    442.1745631...
 20.0    135.3352727...    270.6706113...    593.9941163...
```

This output can then be manipulated using any of MAPLE’s resources for manipulating arrays, but—except by explicit interpolation—it does not have the capacity to generate the solution at values of the independent variable not represented in the output.

• To see the actual coding that MAPLE assembles for the procedure generated by `dsolve`, we must first set the interface variable `verboseproc` to 2 and then execute the command `eval`. For example, the statements

```maple
> interface( verboseproc = 2 ):
> eval( soln );
```

will generate several screens full of MAPLE coding embodying the procedure symbolized by the variable `soln`.

### 11.14 Solving ODEs Numerically with FORTRAN

*Note:* Except for files explicitly flagged as from the Numerical Recipes library, all FORTRAN programs (*.f) and all FORTRAN-created data files (*.f.dat) in this chapter can be copied from the directory `$HEAD/fortran`, where (as defined in the Local Guide) `$HEAD` must be replaced by the appropriate path for your site.

FORTRAN 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.
Table 11.14: First part of FORTRAN coding for decay.f.

```
WRITE(*, '((1X,A))') 'Decay constant for A: ' ! Get parameters
READ(*,*) AK
WRITE(*, '((1X,A))') 'Decay constant for B: ' 
READ(*,*) BK
WRITE(*, '((1X,A))') 'Number of steps : ' 
READ(*,*) NSTEPS
WRITE(*, '((1X,A))') 'Time step : ' ! Get time step
READ(*,*) DT
WRITE(*, '((1X,A))') 'Initial A : ' ! Get initial values
READ(*,*) A
WRITE(*, '((1X,A))') 'Initial B : ' 
READ(*,*) B
WRITE(*, '((1X,A))') 'Initial C : ' 
READ(*,*) C
T = 0.0 ! Set initial time
```

11.14.1 FORTRAN Programs from Scratch

More to illustrate the essence of an algorithm than to develop a truly useful solver, we begin by presenting a quick FORTRAN program to implement Euler’s method as expressed in Eqs. (11.81)–(11.83) 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 listed in Table 11.14.67,68 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

```
OPEN( UNIT = 1, FILE = 'decay_f.dat', STATUS = 'NEW' )
WRITE( 1, 10 ) ! Head table
10 FORMAT( ' T A B C' )
WRITE( 1, 20 ) T, A, B, C ! Output initial values
20 FORMAT( ' ', F6.2, 3F10.3 )
```

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 11.15. A full listing of this program, which is named decay.f, will be found in Section 11.A.

After storing the file containing this program in the default directory, we compile and run it to generate the solution with the statements69

---

67 Remember that, except within quoted strings, FORTRAN is insensitive to case. For the most part, we shall use lower-case letters within textual discussions but upper-case letters in presenting program statements, but we will probably not be entirely consistent.

68 Because we are using implicit data typing, we use $ak$ and $bk$ rather than $ka$ and $kb$ for the decay constants to avoid implying to FORTRAN that these quantities are to be treated as integers.

69 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. 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 11.15: Last lines of FORTRAN coding for decay.f.

```fortran
DO I = 1, NSTEPS
   DADT = -AK*A ! Calculate current derivatives
   DBDT = AK*A - BK*B
   DCDT = BK*B
   A = A + DADT*DT ! Calculate new values
   B = B + DBDT*DT
   C = C + DCDT*DT
   T = T + DT
   WRITE( 1, 20 ) T, A, B, C ! Output new values
ENDDO
CLOSE( UNIT = 1 )
```

f77 -o decay.xf decay.f Create executable file decay.xf.
./decay.xf Execute program.
Decay constant for A: 0.1 Enter controlling values.
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.f.dat, whose first and last few lines will be

```
T    A      B      C
0.00 1000.0 0.000   0.000
0.25 975.0  25.000  0.000
0.50 950.6  48.750  0.625
0.75 926.9  71.297 1.844
49.50 6.651 33.769 959.580
49.75 6.485 33.091 960.424
50.00 6.323 32.426 961.252
```

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.f to use the second-order Runge-Kutta algorithm.

### 11.14.2 Using Numerical Recipes

*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.f*, which advances the solution one step by the fourth-order Runge-Kutta method, and the routine *rkck.f*, which advances the solution one step by the fifth-order Cash-Karp Runge-Kutta method. Other routines in this category, including *mmid.f* (which implements a modified midpoint method), are used in the same way.
One rung up in the Numerical Recipe heirarchy 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 \texttt{rkck.f} is named \texttt{rkqs.f}; the stepper routine that uses \texttt{mmid.f} is \texttt{bsstep.f}.

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 heirarchy. 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.\textsuperscript{70,71}

To use an algorithm or stepper routine, we have two responsibilities. First, we must supply a FORTRAN 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—a vector—returning the derivatives of the dependent variables. Named common (Section 9.5.4) must be used if the problem contains parameters whose values must be communicated to the subroutine. For our standard three-species radioactive decay, for example, that required subroutine might be\textsuperscript{72}

\begin{verbatim}
SUBROUTINE DER( T, AN, DNDT )
DIMENSION AN(*), DNDT(*) ! Take dimensions set in main program
COMMON /PARAMS/ AK, BK ! Locate parameters
DNDT(1) = -AK*AN(1) ! Calculate dA/dt
DNDT(2) = AK*AN(1) - BK*AN(2) ! Calculate dB/dt
DNDT(3) = BK*AN(2) ! Calculate dC/dt
RETURN
END
\end{verbatim}

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 \texttt{rk4.f}, which is called with a statement of the form

\begin{verbatim}
CALL RK4( YS, DYDT, NEQS, T, DT, YF, DRVS )
\end{verbatim}

Here \texttt{YS} and \texttt{DYDT} are vectors containing the values of the dependent variables and their derivatives at time \texttt{T}, \texttt{NEQS} is the number of equations in the system, \texttt{DT} is the amount by which the independent variable is to be advanced, \texttt{YF} is a vector that receives the values of the dependent variables at time \texttt{T+DT}, and \texttt{DRVS}, which is a string value that is \textit{not} enclosed in quotation marks, is the name of the subroutine that \texttt{rk4.f} is to invoke to calculate the derivatives—the subroutine structured as

\textsuperscript{70}Alternatively, we might exploit the driver routines \texttt{rkdumb.f} and \texttt{odeint.f} in the Numerical Recipes library.

\textsuperscript{71}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 \texttt{$NRHEAD/recipes_f DEMO/src$. The sample program corresponding to a particular routine is named by prefixing the character \texttt{x} to the routine name, e.g., \texttt{xrk4.f} for \texttt{rk4.f}.

\textsuperscript{72}Again, to avoid having FORTRAN interpret what should be floating point variables as integers, we name the vector for the dependent variables \texttt{an} and the parameters \texttt{ak} and \texttt{bk}.
described in the last paragraph.\(^{73}\) Note (1) that \texttt{rk4.f} 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 \texttt{rk4.f}; (2) \(y_f\) can be the same vector as \(y_s\), so the output produced by \texttt{rk4.f} can be stored on top of its input; and (3) \texttt{rk4.f} does not increment \(t\), so an explicit incrementation will be necessary before a subsequent call to \texttt{rk4.f} in a loop.

A possible driving program to use subroutine \texttt{der} (which we have written) and subroutine \texttt{rk4.f} (supplied in the \textit{Numerical Recipes} library) would have three major sections. First we dimension variables, assign values to the parameters, and initialize variables with the statements

\begin{verbatim}
DIMENSION AN(3), DNDT(3) ! Dimension vectors
COMMON /PARAMS/ AK, BK ! Locate parameters
AK = 0.1 ! Set parameters
BK = 0.1
DT = 0.25 ! Set time step
AN(1) = 1000.0 ! Initialize dependent variables
AN(2) = 0.0
AN(3) = 0.0
T = 0.0 ! Set initial time
\end{verbatim}

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

\begin{verbatim}
OPEN( UNIT = 1, FILE = 'decayrk4_f.dat', STATUS = 'NEW' )
WRITE( 1, 10 )
10 FORMAT(' T A B C')
WRITE( 1, 20 ) T, AN(1), AN(2), AN(3)
20 FORMAT(' ', F6.2, 3F10.3)
\end{verbatim}

Finally, we execute a loop in which, for each pass, the derivatives at the start of the current interval are evaluated, \texttt{rk4.f} 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

\begin{verbatim}
DO I = 1, 200
   CALL DER( T, AN, DNDT )
   CALL RK4( AN, DNDT, 3, T, DT, AN, DER)
   T = T + DT
   WRITE( 1, 20 ) T, AN(1), AN(2), AN(3)
ENDDO
CLOSE( UNIT = 1 )
\end{verbatim}

A full listing of this program, which is named \texttt{decayrk4.f}, will be found in Section 11.B.

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 \texttt{rk4.f}, we obtain the actual solution and store it in the file \texttt{decayrk4.dat} by

1. Creating the driving program \texttt{decayrk4.f} and the subroutine \texttt{der} using an available text editor (or copying the files from the directory \texttt{$\text{HEAD}/fortran$}).

\(^{73}\) In this generic indication of a calling statement, we deliberately use the argument \texttt{DRVS} rather than \texttt{DER} to emphasize that the statement in this paragraph indeed is generic. Experience has shown that using \texttt{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 \texttt{DER}—an impression that is decidedly incorrect.
2. Copying the recipe \texttt{rk4.f} from the directory \$NRHEAD/recipes.f/recipes to the default directory.

3. Compiling and linking the executable module with the statement

\texttt{f77 -o decayrk4.xf decayrk4.f rk4.f}

4. Running the program with the statement \texttt{./decayrk4.xf}.

The first and last few lines in the file \texttt{decayrk4.f.dat} will be

\begin{verbatim}
 T  A  B  C
 0.00 1000.00 0.000 0.000
 0.25  975.31  24.383  0.307
 0.50  951.23  47.561  1.209
 0.75  927.74  69.581  2.676
 . . . .
49.50  7.083  35.063  957.854
49.75  6.909  34.370  958.722
50.00  6.738  33.690  959.572
\end{verbatim}

We conclude this section by illustrating how to construct a driving program to use the stepper routine \texttt{rkqs}. The derivatives will, of course, be supplied by the same subroutine that we used with \texttt{rk4}. Invocation of \texttt{rkqs}, however, requires a bit more preparation. To call \texttt{rkqs}, we use a statement of the form

\begin{verbatim}
CALL RKQS( Y, DYDT, NEQS, T, DTTRY, RTOL, YS, DTACT, DTNEXT, DRVS )
\end{verbatim}

where \textit{y} and \textit{dydt} are vectors containing the values of the dependent variables and their derivatives at time \textit{t}, \textit{neqs} is the number of equations in the system, \textit{dttry} is the amount by which the independent variable is to be advanced, \textit{rtol} is the desired relative tolerance, \textit{ys} is a vector containing the values relative to which the tolerance is to be assessed, \textit{dtact} is the actual step size on which the adaptive process ultimately settled, \textit{dtnext} is the recommended step size for the next step in the solution, and \textit{drvs} is the name of the subroutine that is to be invoked to calculate the derivatives. Note (1) that \texttt{rkqs.f} requires the current values of the derivatives as \textit{input}, so an explicit call to the subroutine returning those derivatives will have to precede the call to \texttt{rkqs.f}; and (2) in contrast to \texttt{rk4.f}, \texttt{rkqs.f} returns the new values in the variables \textit{y} and \textit{t}, which were used to supply the original values as input.

A driving program to exploit \texttt{rkqs} is similar to—but not identical to—\texttt{decayrk4.f}. Dimensioning variables, setting parameters, and initializing variables is accomplished with the statements listed in Table 11.16. 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

\begin{verbatim}
OPEN( UNIT = 1, FILE = 'decayrkqs_f.dat', STATUS = 'NEW' )
WRITE( 1, 10 )
10 FORMAT(' T  A  B  C')
WRITE( 1, 20 ) T, AN(1), AN(2), AN(3)
20 FORMAT(' ', F6.2, 3F10.3 )
\end{verbatim}

Finally, we execute a loop in which, for each pass, the derivatives at the start of the current interval are evaluated, \texttt{rkqs.f} 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
Table 11.16: First line of FORTRAN coding for decayrkqs.f.

```
DIMENSION AN(3), DNDT(3), ANS(3) ! Dimension vectors
COMMON /PARAMS/ AK, BK ! Locate parameters
AK = 0.1 ! Set parameters
BK = 0.1
DTTRY = 0.25 ! Set time step
AN(1) = 1000.0 ! Initialize dependent variables
AN(2) = 0.0
AN(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
```

DO WHILE (T .LT. 50.0)
    CALL DER( T, AN, DNDT )
    CALL RKQS( AN, DNDT, 3, T, DTTRY, REL, ANS, DTACT,
               DTNEXT, DER )
    WRITE( 1, 20 ) T, AN(1), AN(2), AN(3)
    DTTRY = DTNEXT
ENDDO
CLOSE( UNIT = 1 )
```

A full listing of this program, which is named decayrkqs.f, will be found in Section 11.C.

Starting from scratch with the intention to use rkqs.f, we obtain the actual solution and store it in the file decayrkqs.f.dat by

1. Creating the driving program decayrkqs.f and the subroutine der using an available text editor (or copying the files from the directory $HEAD/fortran$).

2. Copying the recipes rkqs.f and rkck.f (whose importance will be proclaimed if the next step is attempted without it) from the directory $NRHEAD/recipes_f/recipes$ to the default directory.

3. Compiling and linking the executable module with the statement

   ```
f77 -o decayrkqs.xf decayrkqs.f rkqs.f rkck.f
```

4. Running the program with the statement `./decayrkqs.xf`.

The first and last few lines in the file decayrkqs.f.dat will be
Note, in particular, that—because the underlying algorithm is adaptive—this approach has generated solutions at an irregularly spaced sequence of times.

If we deleted the statement \( DTTRY = DTNEXT \) from the loop in `decayrkqs.f`, 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

<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>
</tbody>
</table>

which 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.

### 11.15 Solving ODEs Numerically with C

**Note:** Except for files explicitly flagged as from the Numerical Recipes library, all C programs (`*.c`) and all C-created data files (`*.c.dat`) in this chapter can be copied from the directory `$HEAD/cc`, where (as defined in the Local Guide) `$HEAD` must be replaced by the appropriate path for your site.

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

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. (11.81)–(11.83) 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\(^{74}\)

\(^{74}\)Remember that C is case sensitive.
Table 11.17: Solution loop for decay.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 );
```

```
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

```
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 11.17. 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\textsuperscript{75}

\textsuperscript{75}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. 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.
11.15. SOLVING ODES NUMERICALLY WITH C

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 will be

<table>
<thead>
<tr>
<th>t</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1000.0</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>0.25</td>
<td>975.0</td>
<td>25.000</td>
<td>0.000</td>
</tr>
<tr>
<td>0.50</td>
<td>950.6</td>
<td>48.750</td>
<td>0.625</td>
</tr>
<tr>
<td>0.75</td>
<td>926.9</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>

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 re-casting decay.c to use the second-order Runge-Kutta algorithm.

11.15.2 Using Numerical Recipes

Numerical Recipes provides many interrelated subroutines for solving ODEs. At the bottom of the Numerical Recipe heirarchy 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 heirarchy 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 heirarchy. 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.\textsuperscript{76,77}

\textsuperscript{76}Alternatively, we might exploit the driver routines rkdumb.c and odeint.c in the Numerical Recipes library.

\textsuperscript{77}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.\textsuperscript{78}

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.
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 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). \(^{79}\) 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 variable 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. \(^{80}\) 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

\(^{79}\) Details on the use of global variables in C are discussed in Section 9.7.4.

\(^{80}\) 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.

\(^{81}\) 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]`. 
11.15. SOLVING ODES NUMERICALLY WITH C

311

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, "%.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, "%.2f %.10f %.10f %.10f
", 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.

The first and last few lines in the file decayrk4_c.dat will be

fptr = fopen( "decayrk4_c.dat", "w" );
fprintf( fptr, " t A B C
" );
fprintf( fptr, "%.2f %.10f %.10f %.10f
", 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, "%.2f %.10f %.10f %.10f
", 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.

The first and last few lines in the file decayrk4_c.dat will be
Table 11.18: First statements for C Program decayrkqs.c.

```c
float n[3+1], dnde[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;
```

<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>49.50</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>

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:

```c
rkqs( 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. neqs is the number of equations in the system, dttry is the amount by which the independent variable is to be advanced, rtol is the desired relative tolerance, ys is a vector containing the values relative to which the tolerance is to be assessed, dtact is the actual step size on which the adaptive process ultimately settled, dtnext is the recommended step size for the next step in the solution, and 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 11.18. 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:

---

82 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  313

```c
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] );
```

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

```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` will be

```plaintext
<table>
<thead>
<tr>
<th>t</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1000.00</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>
```
Note, in particular, that—because the underlying algorithm is adaptive—this approach has generated solutions at an irregularly spaced sequence of times.

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

<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>49.75</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>

which 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

$$m \frac{d^2x}{dt^2} + b \frac{dx}{dt} + kx = mf \cos \omega t$$

subject to the general initial conditions

$$x(0) = x_0 \quad ; \quad \frac{dx}{dt}(0) = v_0$$

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 ($m_1$ and $m_2$, 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 on that line, 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

\[
s \left( \frac{A + Bs^2}{\alpha s^4 + \beta s^2 + \gamma} \right) = s \left( \frac{R}{s^2 + R'} + \frac{Q}{s^2 + Q'} \right) = \frac{Rs}{s^2 + R'} + \frac{qs}{s^2 + Q'}
\]

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

\[
m \frac{d^2x_1}{dt^2} = -kx_1 + k(x_2 - x_1)
\]

\[
m \frac{d^2x_2}{dt^2} = -k(x_2 - x_1) + k(x_3 - x_2)
\]

\[
m \frac{d^2x_3}{dt^2} = -k(x_3 - x_2) - kx_3
\]

11.4. The system called the double pendulum shown in Fig. 11.21 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

\[
x(0) = x_0 \quad ; \quad \frac{dx}{dt}(0) = 0
\]

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

\[
\frac{d^2 x}{dt^2} + \omega^2 x = 0
\]

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

\[
x(t) = A \cos \omega t + B \sin \omega t
\]

where \(A\) and \(B\) are constants determined by the initial conditions.

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

\[
m \frac{d^2 \mathbf{r}}{dt^2} = \frac{qQ}{4\pi\varepsilon_0 r^3} \mathbf{r}
\]
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

\[
m\left[\frac{d^2r}{dt^2} - r \left(\frac{d\phi}{dt}\right)^2 \right] = f(r) \quad ; \quad m \left( \frac{d^2\phi}{dt^2} + \frac{2}{r} \frac{dr}{dt} \frac{d\phi}{dt} \right) = 0
\]

(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

\[
mr^2 \frac{d\phi}{dt} = \text{constant} = L \quad \Rightarrow \quad \frac{d\phi}{dt} = \frac{L}{mr^2}
\]

and then that

\[
m \frac{d^2r}{dt^2} = f(r) + \frac{L^2}{mr^3}
\]

Here, \(L = m(x_0v_{x0} - y_0v_{y0})\) 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

\[
r(0) = \sqrt{x_0^2 + y_0^2} \quad \frac{dr}{dt}(0) = \frac{x_0v_{x0} + y_0v_{y0}}{\sqrt{x_0^2 + y_0^2}}
\]

\[
\phi(0) = \arctan \frac{y_0}{x_0} \quad \frac{d\phi}{dt}(0) = \frac{x_0v_{x0} - y_0v_{y0}}{x_0^2 + y_0^2}
\]

(d) Restrict the force to the inverse square gravitational force, finding that

\[
\frac{d^2r}{dt^2} = -\frac{GM}{r^2} + \frac{L^2}{mr^3}
\]

(e) Recast the differential equations and initial conditions in dimensionless form, finding that

\[
\frac{d^2\tau}{d\tau^2} = -\frac{1}{\tau^2} + \frac{\beta}{\tau^3} \quad ; \quad \frac{d\phi}{d\tau} = \pm \sqrt{\beta} \frac{1}{\tau^2}
\]

\[
\tau(0) = \frac{\tau_0}{\ell} = \frac{\sqrt{x_0^2 + y_0^2}}{\ell} \quad ; \quad \phi(0) = \arctan \frac{y_0}{x_0}
\]

\[
\frac{d\tau}{d\tau}(0) = \frac{x_0v_{x0} + y_0v_{y0}}{\sqrt{x_0^2 + y_0^2} \sqrt{GM/\ell}} \quad ; \quad \frac{d\phi}{d\tau}(0) = \frac{x_0v_{x0} - y_0v_{y0}}{(x_0^2 + y_0^2) \sqrt{GM/\ell}}
\]

where \(\beta = L^2/(GMm^2\ell)\) 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 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

\[
\frac{d^2 y}{dx^2} + k^2 y = 0 \quad ; \quad y(0) = y(1) = 0
\]

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

\[
\left. \frac{d^2 y}{dx^2} \right|_{x=x_i} \approx \frac{y_{i+1} - 2y_i + y_{i-1}}{\Delta x^2} \quad ; \quad i = 1, 2, 3, \ldots n - 1
\]

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 \( MY = \alpha Y \), 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.

11.13. `ludiffeq_23` and `ludiffeq_45` allow an additional argument called “report” which instructs IDL to write to the screen the time value, step size, and dependent variable for each row in the data table. This option is selected using the syntax

\[
ludiffeq_23, \ 'function name', \ t, \ x, \ t0=start, \ tf=finish, \ init=ic, \ /report
\]

in the call to `ludiffeq_23` or `ludiffeq_45`. Primarily so that you can see the adaptive nature of the time step of these routines, redo the example of the radioactive decay sequence selecting the `/report` option. Solve the differential equation several times using several different tolerances and notice the effect different tolerances have on the time step. Also, compare the time step used by `ludiffeq_23` and `ludiffeq_45` in solving the same problem to the same tolerance.

11.14. Some radioisotopes exhibit a branched decay sequence such as

```
A ---+--- B
 \   |   /   |
  k1 /  k3
 \   |   |
  k2 /  k4
 C ---+--- D
```

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 \( F = 0 \) to be \( x(0) = 1 \) and \( v(0) = 0 \), produce graphs of \( x \) versus \( t \) for several values of \( \beta \) on the interval \( 0.0 < \beta < 2.0 \). Then using the knowledge that \( \beta = b/m\omega = b/\sqrt{mk} \), and that \( \ell = \omega t \), 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 $\overline{F} = 0$ for the cases of critical damping ($\overline{b} = 2.0$) and overdamping ($\overline{b} > 2.0$).

11.17. Explore the behavior of the Van der Pol oscillator described in dimensionless form by the equation

$$\frac{d^2\pi}{dt^2} = \frac{d\pi}{dt}(1 - \pi^2) - \pi$$

obtaining graphs of position versus time, velocity versus time, and velocity versus position (the phase-plane trajectory), each for several different initial conditions. Convince yourself that the final, steady-state path in the phase plane is independent of the initial conditions.

11.18. The angular position $\theta$ of a simple pendulum of length $l$ satisfies the non-linear equation

$$\frac{d^2\theta}{dt^2} + \frac{g}{l} \sin \theta = 0$$

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 $20^\circ$, $45^\circ$, $90^\circ$, $120^\circ$, $150^\circ$, $165^\circ$, and $178^\circ$. Look particularly at graphs of $\theta$ versus $t$, $d\theta/dt$ versus $t$, and $d\theta/dt$ versus $\theta$ (the phase plot). Obtain also a graph of period versus amplitude (initial displacement). Write several paragraphs describing your setup of the problem and presenting evidence for your discoveries. Optional: Try starting the pendulum at the bottom (0 initial angle) with several 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 $\tilde{t} = \sqrt{g/l} t$ so that the equation becomes $d^2\theta/d\tilde{t}^2 + \sin \theta = 0$.

11.19. 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

$$\frac{d^2\pi}{dt^2} = -\frac{\pi}{(\pi^2 + \overline{\pi}^2)^5}; \quad \frac{d^2\overline{y}}{dt^2} = -\frac{\overline{y}}{(\pi^2 + \overline{\pi}^2)^5}$$

[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.

11.20. 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.

11.21. 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 looks 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 $E = E_x \hat{i}$ and $B = B_z \hat{k}$. In vector form, the equation of motion for this particle is

$$m \frac{d^2\mathbf{r}}{dt^2} = q \mathbf{E} + q \frac{d\mathbf{r}}{dt} \times \mathbf{B}$$

Verify the equations of motion

$$m \frac{d^2x}{dt^2} = qE_x + qB_z \frac{dy}{dt}; \quad m \frac{d^2y}{dt^2} = -qB_x \frac{dx}{dt}; \quad m \frac{d^2z}{dt^2} = 0$$
for the specific fields of this exercise, express them in dimensionless form (note that \( \omega = qB_s/m \) is a frequency and \( E_s/B \) is a velocity), and thoroughly explore the behavior of the particle in this situation. Try to understand the motion intuitively. Hint: You should find that, in terms of an arbitrarily selected unit of length \( \ell \), the equations involve a single parameter \( qE_x/(m\omega^2\ell) \), which can alternatively be written as \( (E_x/B_s)/(\omega\ell) \) — the ratio of the velocity \( E_x/B_s \) 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

\[
\frac{d^2 x}{dt^2} = 0 \quad ; \quad \frac{d^2 y}{dt^2} = -\frac{q \alpha}{m} y = -by
\]

(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 \( y \) 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. (The method required to produce such a plot is discussed in Section 2.16.2.)

11.25. Recast the program \texttt{decay.f} 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.

11.26. 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
\[ \frac{dA}{dt} = -k_f AB + k_r CD \]
\[ \frac{dB}{dt} = -k_f AB + k_r CD \]
\[ \frac{dC}{dt} = k_f AB - k_r CD \]
\[ \frac{dD}{dt} = k_f AB - k_r CD \]

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
\[ \frac{dx}{dt} = -k_1 x + k_2 xy \]
\[ \frac{dy}{dt} = k_3 y - k_4 xy \]

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. Generate a graph analogous to Fig. 11.11 using the improved Euler method and comment on what it reveals about the adequacy of various time steps used.

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.

11.31. Read the book Numerical Recipes to discover how to use the routine odeint and then use it to study three-species radioactive decay.

11.32. Read the book Numerical Recipes to discover how to use the routine mmid and then use it to study three-species radioactive decay.

11.33. Use two different numerical recipes of your choice to study the logistic growth of a population.

11.34. Study the damped, unforced harmonic oscillator using (a) rk4 and (b) rkqs. Consider under-damped, critical, and overdamped cases.

11.35. Use (a) rk4 and (b) rkqs to study both bound and unbound orbits in the inverse square gravitational force.

11.36. Choose one of the exercises from Section 11.16.2 and address it using at least one numerical recipe.
11.A Listing of decay.f

PROGRAM DECAY

WRITE(*, '(1X,A)') 'Decay constant for A: ' ! Get parameters
READ(*,*) AK
WRITE(*, '(1X,A)') 'Decay constant for B: '
READ(*,*) BK
WRITE(*, '(1X,A)') 'Number of steps : '
READ(*,*) NSTEPS
WRITE(*, '(1X,A)') 'Time step : ' ! Get time step
READ(*,*) DT
WRITE(*, '(1X,A)') 'Initial A : ' ! Get initial values
READ(*,*) A
WRITE(*, '(1X,A)') 'Initial B : '
READ(*,*) B
WRITE(*, '(1X,A)') 'Initial C : '
READ(*,*) C
T = 0.0 ! Set initial time

OPEN( UNIT = 1, FILE = 'decay_f.dat', STATUS = 'NEW' )

WRITE( 1, 10 ) ! Head table
10 FORMAT( ' T A B C' )

WRITE( 1, 20 ) T, A, B, C ! Output initial values
20 FORMAT( ' ', F6.2, 3F10.3 )

DO I = 1, NSTEPS
   DADT = -AK*A ! Calculate current derivatives
   DBDT = AK*A - BK*B
   DCDT = BK*B
   A = A + DADT*DT ! Calculate new values
   B = B + DBDT*DT
   C = C + DCDT*DT
   T = T + DT
   WRITE( 1, 20 ) T, A, B, C ! Output new values
ENDDO

CLOSE( UNIT = 1 )

END
11.B Listing of decayrk4.f

PROGRAM DECAYRK4

EXTERNAL DER
DIMENSION AN(3), DNDT(3) ! Dimension vectors

COMMON /PARAMS/ AK, BK ! Locate parameters
AK = 0.1 ! Set parameters
BK = 0.1
DT = 0.25 ! Set time step

AN(1) = 1000.0 ! Initialize dependent variables
AN(2) = 0.0
AN(3) = 0.0
T = 0.0 ! Set initial time

OPEN( UNIT = 1, FILE = 'decayrk4_f.dat', STATUS = 'NEW' )

WRITE( 1, 10 )
10 FORMAT(' T A B C')

WRITE( 1, 20 ) T, AN(1), AN(2), AN(3)
20 FORMAT(' ', F6.2, 3F10.3)

DO I = 1, 200
CALL DER( T, AN, DNDT )
CALL RK4( AN, DNDT, 3, T, DT, AN, DER)
T = T + DT
WRITE( 1, 20 ) T, AN(1), AN(2), AN(3)
ENDDO

CLOSE( UNIT = 1 )

STOP
END

SUBROUTINE DER( T, AN, DNDT )
DIMENSION AN(*), DNDT(*) ! Establish sizes of vectors
COMMON /PARAMS/ AK, BK ! Locate parameters
DNDT(1) = -AK*AN(1) ! Calculate dA/dt
DNDT(2) = AK*AN(1) - BK*AN(2) ! Calculate dB/dt
DNDT(3) = BK*AN(2) ! Calculate dC/dt
RETURN
END
11.C Listing of decayrkqs.f

PROGRAM DECAYRKQS

EXTERNAL DER
DIMENSION AN(3), DNDT(3), ANS(3) ! Dimension vectors

COMMON /PARAMS/ AK, BK ! Locate parameters
AK = 0.1 ! Set parameters
BK = 0.1
DTTRY = 0.25 ! Set trial time step

AN(1) = 1000.0 ! Initialize dependent variables
AN(2) = 0.0
AN(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

OPEN( UNIT = 1, FILE = 'decayrkqs_f.dat', STATUS = 'NEW' )

WRITE( 1, 10 )
10 FORMAT(' T A B C')

WRITE( 1, 20 ) T, AN(1), AN(2), AN(3)
20 FORMAT(' ', F6.2, 3F10.3)

DO WHILE (T .LT. 50.0)
 CALL DER( T, AN, DNDT )
 CALL RKQS( AN, DNDT, 3, T, DTTRY, REL, ANS, DTACT, + DTNEXT, DER )
 WRITE( 1, 20 ) T, AN(1), AN(2), AN(3)
 DTTRY = DTNEXT
 ENDDO

CLOSE( UNIT = 1 )

STOP
END

SUBROUTINE DER( T, AN, DNDT )
DIMENSION AN(*), DNDT(*) ! Establish sizes of vectors
COMMON /PARAMS/ AK, BK ! Locate parameters

DNDT(1) = -AK*AN(1) ! Calculate dA/dt
DNDT(2) = AK*AN(1) - BK*AN(2) ! Calculate dB/dt
DNDT(3) = BK*AN(2) ! Calculate dC/dt
RETURN
END
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\n" );
    fprintf( fptr, "%.2f %.3f %.3f %.3f\n", 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, "%.2f %.3f %.3f %.3f\n", t, A, B, C );
    }

    **** Close file ****/

    fclose( fptr );
}
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;
    t = 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

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

$$\frac{dp}{dt} = f \quad \text{and} \quad \frac{dx}{dt} = v \quad (13.1)$$

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

$$p = \frac{mv}{\sqrt{1 - v^2/c^2}} \quad \text{or} \quad p = mv \quad (13.2)$$
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 \textit{physical} problem of predicting the trajectory reduces to the \textit{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

\[
\frac{d^2 x}{dt^2} = \frac{dv}{dt} = \frac{dv}{dx} \frac{dx}{dt} = v \frac{dv}{dx}
\]

(13.4)

Then, Newton’s second law becomes

\[
m \frac{d^2 x}{dt^2} = f(x) \quad \Rightarrow \quad mv \frac{dv}{dx} = f(x) \quad \Rightarrow \quad mv \, dv = f(x) \, dx
\]

(13.5)

Finally, by integrating this last expression from initial values to general values at some other time, we find that

\[
m \int_{x_0}^{x} v' \, dv' = \frac{1}{2} mv^2 - \frac{1}{2} mv_0^2 = \int_{x_0}^{x} f(x') \, dx'
\]

(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 \( \mathbf{r}_{cm} \) of the \textit{center of mass} of an object is given by

\[
\mathbf{r}_{cm} = \frac{1}{M} \int \mathbf{r} \, dm
\]

(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

\[
\mathbf{r}_{cm} = \frac{1}{M} \int \sigma(r, \phi) \, r \, dr \, d\phi = \frac{1}{M} \int \sigma(r, \phi) \left( r \cos \phi \ \mathbf{i} + r \sin \phi \ \mathbf{j} \right) \, r \, dr \, d\phi
\]

(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.
Figure 13.1: A semicircular disk.

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 \(\sigma = M/(\frac{1}{2}\pi a^2)\) shown in Fig. 13.1, we would conclude that

\[
r_{cm} = \frac{1}{M} \int_0^a \int_0^{\pi} \frac{M}{\frac{1}{2}\pi a^2} \left( r \cos \phi \hat{i} + r \sin \phi \hat{j} \right) r \, dr \, d\phi = \frac{2}{\pi a^2} \int_0^a \int_0^{\pi} r^2 \left( \cos \phi \hat{i} + \sin \phi \hat{j} \right) dr \, d\phi
\]

(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

\[
r_{cm} = \frac{2}{\pi} \int_0^1 \int_0^{\pi} \lambda^2 \left( \cos \phi \hat{i} + \sin \phi \hat{j} \right) d\phi \, d\lambda
\]

(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

\[
I = \int r^2 \, dm
\]

(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

\[
I = Mk^2 \quad \Rightarrow \quad k = \sqrt{\frac{I}{M}}
\]

(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 \(\sigma = M/(\frac{1}{2}\pi a^2)\). 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

\[
I = \int_{y=0}^{y=a} y^2 \, dm
\]

(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

$$dm = \sigma dA = \frac{4M}{\pi a^2} \sqrt{a^2 - y^2} dy$$

Finally, the moment of inertia of the entire object is given by the integral

$$I = \frac{4M}{\pi a^2} \int_0^a y^2 \sqrt{a^2 - y^2} dy = \frac{4}{\pi} Ma^2 \int_0^1 \lambda^2 \sqrt{1 - \lambda^2} d\lambda$$

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

$$\text{energy} = \frac{1}{2} I \omega^2 + mgy = \frac{1}{2} ml^2 \left( \frac{d\theta}{dt} \right)^2 - mgl \cos \theta$$

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

$$\frac{1}{2} ml^2 \left( \frac{d\theta}{dt} \right)^2 - mgl \cos \theta = -mgl \cos \theta_0 \quad \Rightarrow \quad \frac{d\theta}{dt} = \pm \sqrt{\frac{2g}{l}} \sqrt{\cos \theta - \cos \theta_0}$$

(We assume that the pendulum does not have sufficient energy to swing over the top.) This last relationship then leads to the conclusion that

$$dt = \pm \sqrt{\frac{l}{2g}} \frac{d\theta}{\sqrt{\cos \theta - \cos \theta_0}} \quad \Rightarrow \quad \int_0^{T/4} dt = \frac{T}{4} = \sqrt{\frac{l}{2g}} \int_0^{\theta_0} \frac{d\theta}{\sqrt{\cos \theta - \cos \theta_0}}$$
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

\[
T = 2\sqrt{\frac{l}{g}} \int_{0}^{\theta_0} \frac{d\theta}{\sqrt{\sin^2(\theta_0/2) - \sin^2(\theta/2)}}
\]  

(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 \(T_0 = 2\pi\sqrt{lg}\), 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

\[
\frac{T(k)}{T_0} = 2 \pi \int_{0}^{\pi/2} \frac{d\phi}{\sqrt{1 - k^2 \sin^2 \phi}} = 2 \pi K(k)
\]  

(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

\[
G(x) = \frac{1}{\sqrt{2\pi} \sigma^2} e^{-(x-\mu)^2/(2\sigma^2)}
\]  

(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

\[
C(u) = \int_0^u \cos \left( \frac{\pi t^2}{2} \right) \, dt \quad \text{;} \quad S(u) = \int_0^u \sin \left( \frac{\pi t^2}{2} \right) \, dt \quad (13.27)
\]

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 their centers separated by \( 2a \) 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 \( r = r \cos \phi \hat{i} + r \sin \phi \hat{j} + z \hat{k} \), produced by one such ring positioned in the \( xy \) plane with its center at the origin. Let \( r' = a \cos \phi' \hat{i} + a \sin \phi' \hat{j} \) locate an element on that (single) ring. In this notation,

\[
r - r' = (a \cos \phi - a \cos \phi') \hat{i} + (a \sin \phi - a \sin \phi') \hat{j} + z \hat{k}
\]

so

\[
|r - r'| = \left[ (r \cos \phi - a \cos \phi')^2 + (r \sin \phi - a \sin \phi')^2 + z^2 \right]^{1/2}
\]

\[
= \left[ r^2 + a^2 - 2ar \cos (\phi' - \phi) + z^2 \right]^{1/2}
\]

Placing these results into the second member of Eq. (13.28) and recognizing that \( dq' = \lambda a \, d\phi' \), we at last find that

\[
V_{\text{one}}(r, \phi, z) = \frac{1}{4\pi \epsilon_0} \int_0^{2\pi} \frac{\lambda a \, d\phi'}{\left[ r^2 + a^2 - 2ar \cos (\phi' - \phi) + z^2 \right]^{1/2}} \quad (13.32)
\]
The entire integral assumes a simpler appearance, however, on the variable \( \alpha = \phi' - \phi \), becoming

\[
V_{\text{one}}(r, \phi, z) = \frac{\lambda a}{4\pi\epsilon_0} \int_{0}^{2\pi} \frac{d\alpha}{[r^2 + a^2 - 2ar \cos \alpha + z^2]^{1/2}}
\]  

(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 2\( a \). 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

\[
V_{\text{midplane}}(r) = V_{\text{one}}(r, \phi, -a) + V_{\text{one}}(r, \phi, a) = \frac{2\lambda a}{4\pi\epsilon_0} \int_{0}^{2\pi} \frac{d\alpha}{[r^2 + 2a^2 - 2ar \cos \alpha + 2]^{1/2}}
\]  

(13.34)

Finally, introducing the variable \( s \) defined by \( r = sa \) to express the radial coordinate in dimensionless terms, we find the expression

\[
V_{\text{two}}(s) = V_{\text{midplane}}(sa) = \frac{2\lambda}{4\pi\epsilon_0 a} \int_{0}^{2\pi} \frac{d\alpha}{[s^2 - 2s \cos \alpha + 2]^{1/2}}
\]  

(13.35)

or, even better, the expression

\[
V(s) = \frac{\sqrt{2}}{2\pi} \int_{0}^{2\pi} \frac{d\alpha}{[s^2 - 2s \cos \alpha + 2]^{1/2}}
\]  

(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 2\( a \). 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

$$P(x_1 < x < x_2) = \int_{x_1}^{x_2} |\psi(x)|^2 \, dx$$  \hspace{1cm} (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

$$\psi(x) = \left( \frac{m\omega}{\pi \hbar} \right)^{1/4} e^{-y^2/2}$$  \hspace{1cm} (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

$$\frac{1}{2} \hbar \omega = \frac{1}{2} k x_{\text{turn}}^2 \implies x_{\text{turn}} = \sqrt{\frac{\hbar \omega}{k}} = a$$  \hspace{1cm} (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

$$P\left(|x| > |x_{\text{turn}}|\right) = \int_{-\infty}^{-a} |\psi(x)|^2 \, dx + \int_{a}^{\infty} |\psi(x)|^2 \, dx = 1 - \int_{-a}^{a} |\psi(x)|^2 \, dx$$  \hspace{1cm} (13.40)

Finally, after substituting the wave function and recasting the entire integral as an integral on the variable $y$, we find that

$$P\left(|x| > |x_{\text{turn}}|\right) = 1 - \frac{1}{\sqrt{\pi}} \int_{-1}^{1} e^{-y^2} \, dy = 1 - \frac{2}{\sqrt{\pi}} \int_{0}^{1} e^{-y^2} \, dy = 1 - \text{erf}(1)$$  \hspace{1cm} (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

$$\int_{a}^{b} \phi_i(x) \phi_j(x) \, w(x) \, dx = N_j \delta_{ij}$$  \hspace{1cm} (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

$$f(x) = \sum_{n} a_n \phi_n(x)$$  \hspace{1cm} (13.43)
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

$$\int_a^b w(x) \phi_j(x) f(x) \, dx = \sum_n a_n \int_a^b w(x) \phi_j(x) \phi_n(x) \, dx = \sum_n a_n N_n \delta_{nj} = N_j a_j$$

We conclude that, if we know either of $f(x)$ or $a_n$, we can determine the other, i.e., that

$$f(x) = \sum_n a_n \phi_n(x) \quad \Longleftrightarrow \quad a_n = \frac{1}{N_n} \int_a^b w(x) \phi_n(x) f(x) \, dx \quad (13.45)$$

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
given by

$$\{1, \sin \frac{n \pi x}{l}, \cos \frac{n \pi x}{l} ; \quad n, m = 1, 2, 3, \ldots \} \quad (13.46)$$

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

$$a_n = \frac{1}{l} \int_{-l}^l f(x) \cos \frac{n \pi x}{l} \, dx \quad ; \quad b_n = \frac{1}{l} \int_{-l}^l f(x) \sin \frac{n \pi x}{l} \, dx \quad (13.51)$$

### 13.3 Evaluating Integrals Symbolically with MAPLE

The MAPLE command for symbolic integration is \texttt{int}, though the synonym \texttt{integrate} will also work. The inert command \texttt{Int}, which postpones explicit evaluation, also exists. The first argument in these commands either is or stands for the integrand, and the second argument specifies the integration variable (with an optional range to convey limits). Thus, for example, the statement

\begin{verbatim}
int( sin(k*x), x );
\end{verbatim}
evaluates the *indefinite* integral $\int \sin(kx) \, dx$—though does not include the arbitrary integration constant—while the statement

\[
\text{int}( \sin(kx), x = 0..\Pi );
\]

evaluates the *definite* integral $\int_0^\pi \sin(kx) \, dx$. If MAPLE is unable to evaluate a particular integral, the program returns the noun form of the integral, i.e., the program simply returns the integral that was provided as input. MAPLE also evaluates improper integrals, recognizing the symbols $\infty$ for $\infty$ and $-\infty$ for $-\infty$. Whether integrating by hand or with the help of a symbolic program, we must always be wary of integrals when the integrand has a singularity in the interval of integration. Indeed, MAPLE checks for these singularities automatically unless the option `$\text{continuous}$', which informs MAPLE that the integrand has no singularities in the interval of integration, has been specified as an argument.

In this section, we illustrate the use of the command `int` to evaluate some of the integrals deduced in Section 13.1. Beyond integration per se, we also show how, in many cases, MAPLE can be used to set up the integral as well. To abbreviate the presentation of MAPLE dialogs, we make liberal use of terminating colons to suppress intermediate output. You are therefore urged to duplicate the dialogs in an actual session with MAPLE, replacing colons with semicolons.

### 13.3.1 Relativistic Motion Under a Constant Force

Suppose the particle to which Eq. (13.3) applies moves relativistically, starting from rest at the origin $[x(0) = 0, v(0) = 0, p(0) = 0]$, and experiences a *constant force* $f$. Then, the basic relationships from which we would determine $x(t)$ and $v(t)$ are

\[
\begin{align*}
p(t) &= \int_0^t f \, dt', \\
x(t) &= \int_0^t v(t') \, dt', \quad \text{and} \quad p(t) = \frac{m v(t)}{\sqrt{1 - v(t)^2/c^2}} \tag{13.52}
\end{align*}
\]

The “conversation” with MAPLE listed in Table 13.1 will find the desired quantities and explore a few of their properties.

Reassuringly, the velocity approaches $c$ as $t \to \infty$ and, with the acceleration $a$ identified as $f/m$, $x$ and $v$ for small $t$ (classical limit) are $\frac{1}{2} at^2$ and $at$ respectively.

### 13.3.2 Center of Mass

We next find the center of mass of the uniform, semicircular plate described in Section 13.1.2 as given by Eq. (13.9). We evaluate this integral with the statements

\[
\begin{align*}
> & \text{with}( \text{LinearAlgebra} ); \quad \text{Add package LinearAlgebra}. \\
> & \text{rr := Vector}( [r*\cos(\phi), r*\sin(\phi), 0] ); \quad \text{Assign r}. \\
> & \text{sigma := m} / (\Pi * a^2 / 2); \quad \text{Evaluate } \sigma, \text{ which is constant}.
\end{align*}
\]
Table 13.1: MAPLE code for relativistic motion under constant force.

```maple
> interface( showassumed=0 );
Suppress display of assumptions.
> assume( c>0, f>0, m>0, t>0 );
Assume c, f, m, and t to be positive.
> p := int(f, tp=0..t);
Integrate (constant) f to find p.

p := ft

> p = m * v / sqrt(1 - v^2/c^2);
Relate momentum to velocity.

ft = mv/√(1 - v^2/c^2)

> soln := solve( %, v );
Extract first solution. (Physically, v and f
must have the same sign.)

soln := ftc

> v := soln[1];
Find limit of v as t → ∞.

ftc

> limit( v, t=infinity );
Substitute tp for t in v.

c

> vp := subs( t=tp, v );
Integrate vp to find position.

> x := int( vp, tp=0..t );

x := c(-cm + √(t^2f^2 + m^2c^2))

> vclassical := taylor( v, t=0, 4 );
Find limit of v for small t.

vclassical := f m t - 1/2 c^2 m^3 t^3 + O(t^5)

> xclassical := taylor( x, t=0, 5 );
Find limit of x for small t.

xclassical := 1/2 f m t^2 - 1/8 c^2 m^3 t^4 + O(t^6)

> restart:
```

---

```maple
> integ := ScalarMultiply( rr, sigma*r );
Evaluate integrand, using ScalarMultiply
from the package LinearAlgebra.

> fst_int := map( int, integ, phi=0..Pi );
Evaluate integral on φ.

> scnd_int := map( int, fst_int, r=0..a );
Evaluate integral on r.

> r_cm := ScalarMultiply( scnd_int, 1/m );
Multiply each component by 1/m.

r_cm :=

```

```
```

```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
Since only the $y$ component of $\mathbf{r}_{\text{cm}}$ differs from zero, the center of mass lies on the $y$ axis—certainly not a surprise—a fraction $4/3\pi = 0.4244$ of the radius from the center of the semicircle. Note that, to effect an element-by-element integration of the components of the vector, we have had to use MAPLE’s command `map` to apply the command `int` to each element of the vector in turn.

Note also that we have evaluated the double integral as a sequence of single integrals. In the present case, the two integrals are independent of one another, and the order in which we perform the integrals is irrelevant. In some cases, however, the order may be important—as, for example, when the limits on one integration variable happen to depend on the other integration variable.

### 13.3.3 Moment of Inertia; Radius of Gyration

The evaluation by MAPLE of the moment of inertia of a semicircular plate as given by Eq. (13.16) is quick and straightforward. We need only the statements\(^3\)

$$
\begin{align*}
&\text{\texttt{\textgreater \ int( lambda^2*sqrt(1-lambda^2), lambda=0..1 )};} \\
&\frac{1}{16}\pi \\
&\text{\texttt{II := m*a^2*(4/Pi)*%;}} \\
&II := \frac{1}{4}ma^2 \\
&\text{\texttt{k := simplify( sqrt(II/m), assume=positive );}} \\
&k := \frac{1}{2}a \\
&\text{\texttt{\textgreater \ restart:}}
\end{align*}
$$

\(^3\)The variable $I$ in MAPLE is reserved for $\sqrt{-1}$, so we use $II$ for the moment of inertia.

to evaluate the necessary integral and find both the moment of inertia $I = ma^2/4$ and the radius of gyration $k = a/2$.

### 13.3.4 Electrostatic Potential of a Finite Line Charge

For a fourth example, we evaluate the electrostatic potential of a uniformly charged finite line extending along the $z$ axis over the interval $-a \leq z \leq +a$. The geometry is shown in Fig. 13.4. With $\lambda$ representing the linear charge density on the line, $dz'$ giving the length of an element of the line, $\mathbf{r}$ and $\mathbf{r}'$ locating the observation point and an element of the source, respectively, and $dq' = \lambda dz'$, we deduce from Eq. (13.28) that the potential established by this source is given by

$$
V(x, y, z) = \frac{\lambda}{4\pi\epsilon_0} \int_{-a}^{+a} \frac{dz'}{|\mathbf{r} - \mathbf{r}'|}
$$

Electing cylindrical coordinates $(r, \phi, z)$, we set $\mathbf{r}' = z'\hat{k}$ and $\mathbf{r} = r\cos\phi\hat{i} + r\sin\phi\hat{j} + z\hat{k}$. Then we construct the integrand and evaluate the integral with the statements

$$
\begin{align*}
&\text{\texttt{\textgreater \ with(LinearAlgebra):}} \\
&\text{\texttt{\rightarrow rr := < r*cos(phi) | r*sin(phi) | z >:}} \\
&\text{\texttt{\rightarrow rp := < 0 | 0 | zp >:}} \\
&\text{\texttt{\rightarrow sep := MatrixAdd( rr, rp, 1, -1 ):}} \\
&\text{\texttt{\rightarrow DotProduct( sep, sep, conjugate=false ):}}
\end{align*}
$$
13.3. EVALUATING INTEGRALS SYMBOLICALLY WITH MAPLE

Figure 13.4: A line charge lying on the $z$ axis.

> simplify( % );  
Recognize $\cos^2 \phi + \sin^2 \phi = 1$.  

> den := sqrt( % );  
Bind $|r - r'|$ to $\text{den}$.  

> cnst := 1/(4*Pi*epsilon[0]);  
Bind $1/4\pi \epsilon_0$ to $\text{cnst}$.  

> integ := lambda * cnst / den;  
Set integrand.  

$$\text{integ} := \frac{\lambda}{4 \pi \epsilon_0 \sqrt{r^2 + z^2 - 2zz' + z'^2}}$$

> interface( showassumed=0 );  
Suppress display of assumptions.  

> assume( r>0, a>0 );  
Assume $r$ and $a$ to be positive.  

> V := int( integ, zp=-a..a );  
Find $V(r, z)$, with $r = \sqrt{x^2 + y^2}$.  

Some versions of MAPLE return the value

$$V := \frac{\lambda}{4 \pi \epsilon_0} \left( \text{arcsinh}\left(\frac{-z + a}{r}\right) + \text{arcsinh}\left(\frac{z + a}{r}\right) \right)$$

for $V$ while others return the equivalent value

$$V := -\frac{\lambda}{4 \pi \epsilon_0} \left( \ln(-z - a + \sqrt{z^2 + a^2 + 2az + r^2}) - \ln(-z + a + \sqrt{z^2 + a^2 - 2az + r^2}) \right)$$

While correct, neither of these results is particularly transparent. Let us test either by examining its behavior in the $xy$ plane in the two limits $a/r \ll 1$ and $a/r \gg 1$, i.e., when $r$ is large compared to the length of the line and when $r$ is small compared to the length of the line. Conveniently, the same sequence of statements appears to work successfully with either form, though the intermediate results (here suppressed) will differ. For the first limit, we execute the statements

> V0 := eval( V, z = 0 );  
Evaluate potential in $xy$ plane.  

> subs( a=s*r, V0 );  
Let $a/r \to s$.  

> taylor( %, s=0, 2 );  
Expand near $s = 0$ ($a/r \to 0, r \to \infty$).  

> subs( s=a/r, % );  
Return to original variable.  

> subs( lambda=q/(2*a), % );  
Express in terms of total charge.

$$\frac{q}{4\pi \epsilon_0 r} + O\left(\frac{a^3}{r^3}\right)$$
As expected, the potential at points remote from the charged line varies with distance like that of a point charge. From a point far away from the wire (compared to its length), the wire looks like a point charge.

At the other extreme, when \( r \) is small compared to the length of the line, we would expand and simplify the potential with the statements

\[
V_1 := \text{series}( V_0, r=0, 2 ) : \quad \text{Expand potential at } z = 0 \text{ for small } r, \text{ keeping only first non-zero term.}
\]

\[
V_2 := \text{convert}( V_1, \text{polynom} ) ; \quad \text{Convert series to a polynomial.}
\]

The result obtained for \( V_2 \) depends on the version of MAPLE and may also be context and platform dependent. For the first value of \( V \) above, MAPLE may return

\[
V_2 := \frac{\lambda}{2\pi \epsilon_0} \left( -\ln \left( \frac{1}{2a} \right) - \ln(r) \right)
\]

while, for the second value of \( V \), MAPLE returns

\[
V_2 := -\frac{\lambda}{4\pi \epsilon_0} \left( \ln \left( \frac{1}{2a} \right) + 2 \ln(r) - \ln(2a) \right)
\]

Whatever is returned, the results will be equivalent to these two which are, of course, equivalent to each other. An even simpler expression results from a judicious use of \text{combine}. For the first form, the statement \text{combine}(2*V2)/2 yields

\[
\frac{\lambda}{2\pi \epsilon_0} \ln \left( \frac{2a}{r} \right)
\]

while for the second form, the statement \text{combine}(4*V2)/4 yields

\[
\frac{\lambda}{4\pi \epsilon_0} \ln \left( \frac{4a^2}{r^2} \right)
\]

though even these results may be version, context, and platform dependent. Though it may be difficult to persuade MAPLE to produce this form, these two results are equivalent to

\[
-\frac{\lambda}{2\pi \epsilon_0} \ln \frac{r}{a} + \frac{\lambda \ln 2}{2\pi \epsilon_0}
\]

which differs by a constant from the familiar logarithmic result for the potential of a uniformly charged, infinitely long straight wire when the reference point (point of zero potential) is taken at a distance \( a \) from the wire. From a point close to the wire (compared to its length) and near its center, the wire looks to be infinitely long.

Before going on to the next section, you should exit from and start a fresh session with MAPLE or, alternatively, issue a \text{restart:} command.

### 13.3.5 The Helmholtz Coil

Next, we evaluate the magnetic field produced on the \( z \) axis by a pair of circular current loops of arbitrary separation and demonstrate the significance of the specific separation used in the Helmholtz coil. Strategically, we regard the pair as a superposition of two loops and seek first the magnetic field of a \text{single} loop. In general, the magnetic field \( \mathbf{B}(\mathbf{r}) \) is given by the Biot-Savart law, which is the first member of Eq. (13.29). If the (single) loop has radius \( a \) and lies in the \( xy \) plane with its center at the origin, then, for an observation point on the \( z \) axis, the various vectors in the expression are given by \( \mathbf{r} = sa \hat{k} \), where (to facilitate expressing things in dimensionless form) the \( z \) coordinate is written as a multiple \( s \) of the radius \( a \) of the loop, \( \mathbf{r}' = a \cos \phi \hat{i} + a \sin \phi \hat{j} \), and \( d\mathbf{r}' = (-a \sin \phi \, \hat{i} + a \cos \phi \, \hat{j}) \, d\phi \). Evaluation of the integrand and the integral for a single loop then proceeds with the statements pp
13.3. EVALUATING INTEGRALS SYMBOLICALLY WITH MAPLE

Suppress display of assumptions.
Assume \( a, \phi, \) and \( s \) to be positive.
Assign \( r \) as a vector.
Assign \( r' \) as a vector.
Evaluate \( r - r' \).
Evaluate \( |r - r'|^2 \).
Recognize \( \cos^2 \phi + \sin^2 \phi = 1 \); set \( \text{mag2} = |r - r'|^2 \).
Recognize \( \cos^2 \phi + \sin^2 \phi = 1 \).

Finally, we bring various factors together to construct the integrand, simplify that integrand, evaluate the integral, and recast the results in a dimensionless form with the statements

\[
\int \text{integ} = \text{LinearAlgebra}[\text{ScalarMultiply}](\text{num}, \mu_0i/(4\pi\text{mag2}^{3/2}))
\]

\[
\text{integ} := \text{map}(\text{simplify}, \%)
\]

\[
\begin{align*}
\mu_0i \cos(\phi) s & \quad \mu_0i \sin(\phi) s & \quad \mu_0i \\
4\pi a(1 + s^2)^{3/2} & \quad 4\pi a(1 + s^2)^{3/2} & \quad 4\pi a(1 + s^2)^{3/2}
\end{align*}
\]

\[
\text{b1} := \text{map}(\text{int}, \%, \phi=0..2\pi)
\]

\[
\text{b1} := \left[ 0, 0, \frac{\mu_0i}{2a(1 + s^2)^{3/2}} \right]
\]

\[
\text{b1z} := \text{b1}[3]
\]

\[
\text{b1c} := \text{eval}(\text{b1z}, s = 0)
\]

\[
\text{b1z} := \text{b1z} / \text{b1c}
\]

\[
\text{b1z} := \frac{1}{(1 + s^2)^{3/2}}
\]

At the beginning, we were interested not in the field of a single loop but in the field of a pair of loops. Thus, we next combine the fields of two separate loops, one positioned a distance \( ca \) above the midplane and the other positioned a distance \( ca \) below the midplane, and again normalize the field so that it is measured in units defined by its value at the origin midway between the two loops. These objectives are accomplished with the statements

\[
\text{bpair} := \text{subs}(s=s-c, \text{b1z}) + \text{subs}(s=s+c, \text{b1z})
\]

\[
\text{bpair} := \text{eval}(\text{bpair}, s = 0)
\]

\[
\text{bpair} := \text{bpair} / \text{bpairc}
\]

\[
\text{bpair} := \frac{1}{2} \left( \frac{1}{1 + (s - c)^2}^{3/2} + \frac{1}{1 + (s + c)^2}^{3/2} \right)(1 + c^2)^{3/2}
\]

This expression is complicated. We explore it further in two ways. First, let us plot a few graphs of this result as a function of \( s \) for representative values of \( c \) with the statements
Figure 13.5: On-axis magnetic field of a pair of loops. The highest of the three graphs shows the field when the loop separation is $c = 1.0$ (separation of the two loops equal to twice the radius of each loop). The middle graph corresponds to $c = 0.5$ (the Helmholtz separation) and the lower graph corresponds to $c = 0.25$.

The resulting graph is shown in Fig. 13.5. As will be further supported in the next paragraph, the middle of the three graphs, which corresponds to $c = 0.5$ (separation of coils equal to the radius of each coil), has the largest region of near constant field near the on-axis point midway between the current loops (i.e., near $s = 0$).

Second, let us examine the behavior of the field near the center ($s = 0$) more closely. To do so, we suppose $s \to 0$ and look at the Taylor expansion of the field about $s = 0$ using the statement

\[
\begin{align*}
\text{tay} & := \text{simplify( taylor( bpair, s=0, 6 ) );} \\
& := 1 + \frac{3}{2} \left( \frac{-1 + 4c^2}{(1 + c^2)^2} \right) s^2 + \frac{15}{8} \left( \frac{1 - 12c^2 + 8c^4}{(1 + c^2)^4} \right) s^4 + O(s^6)
\end{align*}
\]

Clearly there is a particular value of $c$—half the loop separation—at which the coefficient of the $s^2$ term is zero! For that separation, the magnetic field near the center of the loops is especially constant. We find that special value of $c$ with the statements\(^4\)

\[^4\text{Remember the structure of the storage of Taylor series as described in Section 7.8.7.}\]
13.3. EVALUATING INTEGRALS SYMBOLICALLY WITH MAPLE

> op( 2, op(3,%) );  
$-1 + 4c^2$  
Extract coefficient of numerator of second term of Taylor series.

> solve( %=0, c );  
$1/2, -1/2$  
Find $c$ to make numerator zero.

The first of these statements extracts the numerator of the second term of the Taylor series and the second finds the value of $c$ that makes that numerator zero. The two solutions are equivalent. Both reveal that the physical arrangement in which the field has no $s^2$ term has one loop positioned one-half of the loop radius above the midplane and the other positioned one-half of the loop radius below the midplane. That is, the loops are separated by their common radius—the Helmholtz separation.

When the loops have that special separation, we determine numerically how the field varies with $s$ by using the statements

> tay := convert( tay, polynom );  
$\frac{1}{2} + \frac{3}{2} (-1 + 4c^2) s^2 + \frac{15}{8} (1 - 12c^2 + 8c^4) s^4$  
Convert Taylor series to polynomial.

> eval( tay, c = 1/2 );  
$1 - \frac{144}{125}s^4$  
Evaluate field for Helmholtz separation.

> tay1 := evalf( % );  
$tay1 := 1. - 1.152000000s^4$  
Cast result in floating form.

and we see again that, so long as we don’t stray too far from the center of the arrangement, the magnetic field falls away from its central value as the fourth power of the distance from the center. In particular (see next paragraph for a refinement), we might determine approximately how far we can stray from the center along the axis before the field has fallen to, say, 99% or 95% of its value at the center by executing the statements\(^5\)

> solve( tay1=0.99, s );  
$..., 0.3052367918,...$  
Find $s$ when field is down 1%.

> solve( tay1=0.95, s );  
$..., 0.4564354646,...$  
Find $s$ when field is down 5%.

Unfortunately, these solutions are only approximate, since we have used a truncated expansion of the field, an expansion that becomes increasingly incorrect as $s$ increases.

To find more accurately how far from the center we can stray before the field has fallen to 99% or 95% of its value at the center, we need return to the exact expression of \( bpair \) but restrict it to the Helmholtz case by setting $c = 1/2$ with the statement

> bhelm := eval( bpair, c = 1/2 );

Then, we seek the values of $s$ at which this expression assumes the value 0.99 (1% fall off) or 0.95 (5% fall off). Equivalently, we seek the roots of the expressions $bhelm - 0.99 = 0$ and $bhelm - 0.95 = 0$. The dependence of these expressions on $s$, however, is complicated, and the command \texttt{solve} is not up to the task. Thus, we draw (prematurely—see Section 14.12) on MAPLE’s command \texttt{fsolve}, which uses an iterative numerical method to seek a root in a supplied range of values. The above results provide us with sensible stipulations for that range, so we invoke the statements\(^6\)

\(^{5}\)Since we are asking for the solution to a fourth order equation, there will be four solutions. MAPLE will find all of them. In the output, we suppress all but the relevant solution.

\(^{6}\)In \texttt{fsolve}, the first argument is the equation whose root is sought, the second is the variable to be adjusted in finding the root, and the third is the range on that variable in which a root should be sought. By default, \texttt{fsolve} strives to find the root to an absolute tolerance of 10 significant digits.
> fsolve( bhelm - 0.99 = 0, s, 0.2..0.4 );  
0.3137460252  
> fsolve( bhelm - 0.95 = 0, s, 0.3..0.6 );  
0.4884550124  
Finally, out of curiosity, we ask about the value of the magnetic field at a point in the center of one of the coils by executing the statement

> evalf( eval( bhelm, s=0.5 ) );  
0.9458241850  
> restart:  

In summary, as \( s \) increases (i.e., as the observation point moves away from the center of the Helmholtz pair), the on-axis field falls away from its value at the center, at least initially, by an amount proportional to \( s^4 = \left(\frac{z}{a}\right)^4 \). As the above results reveal, we would have to move 31.4\% of the radius (62.8\% of the distance from the center to the plane of either loop) before the field has fallen to 99\% of its value at the center, and 48.8\% of the radius (97.6\% of the distance from the center to the plane of either loop) before the field has fallen to 95\% of its value at the center. By the time we reach the plane of either loop, the field has fallen only to 94.6\% of its value at the center. This optimal configuration of a pair of current loops is regularly used to produce a nearly uniform field over a large volume of space.

### 13.3.6 Period of a Pendulum: Correction as Amplitude Grows

The integral in Eq. (13.22) does not have a simple or familiar evaluation in closed form. MAPLE does, however, make it easy to examine the way in which the period departs from its limiting value as the amplitude moves away from the very small. We simply determine a Taylor expansion of the integrand with the statements

> integ := 1/sqrt(1-k^2*sin(phi)^2);  
\( \integ := \frac{1}{\sqrt{1 - k^2 \sin^2 \phi}} \)

> intapprox := taylor( integ, k=0, 6 );  
\( \text{intapprox} := 1 + \frac{1}{2} (\sin^2 \phi) k^2 + \frac{3}{8} (\sin^4 \phi) k^4 + O(k^6) \)

Then we insert the pre-multiplying factor \( 2/\pi \) and integrate the series on \( \phi \) with the statement

> (2*int( intapprox, phi=0..Pi/2 ) / Pi );  
> expand( convert( %, polynom ) );  
\( 1 + \frac{1}{4} k^2 + \frac{9}{64} k^4 \)

Finally, we substitute \( \sin(\theta_0/2) \) for \( k \) with the statement

> subs(k=sin(theta0/2), %);  
\( 1 + \frac{1}{4} \sin^2 \left(\frac{\theta_0}{2}\right) + \frac{9}{64} \sin^4 \left(\frac{\theta_0}{2}\right) \)

and expand this result as a power series in \( \theta_0 \)
> taylor( %, theta0=0, 6 );
\[ 1 + \frac{1}{16} \theta_0^2 + \frac{11}{3072} \theta_0^4 + O(\theta_0^6) \]

Note that $\theta_0$ must here be expressed in radians.

To see how significantly the period departs from the small-amplitude value when $\theta_0$ is not quite zero, let us evaluate this expression for $\theta_0 = 5^\circ$, $10^\circ$, $15^\circ$, and $20^\circ$. We use the MAPLE statements

> deg := < 5 | 10 | 15 | 20 >:
> ampl := LinearAlgebra[ScalarMultiply]( deg, Pi/180 ):
> per1 := Array(1..4):
> for i from 1 to 4 do per1[i] := 1+ampl[i]^2/16: end do:
> map( evalf, per1 );
\[ [1.000475965, 1.001903859, 1.004283682, 1.007615435] \]

and add the second correction with the statements

> for i from 1 to 4 do per1[i] := per1[i]+ 11*ampl[i]^4/3072: end do:
> map( evalf, per1 );
\[ [1.000476173, 1.001907182, 1.004300503, 1.007668597] \]

Evidently, at amplitudes of $5^\circ$, $10^\circ$, $15^\circ$, and $20^\circ$, the period of the pendulum is approximately $0.05\%$, $0.2\%$, $0.4\%$, and $0.8\%$ larger than the standard small-amplitude approximation given by the expression $2\pi \sqrt{l/g}$.

### 13.3.7 Fourier Coefficients for Half-Rectified Signal

Suppose we sought to express the half-rectified sine wave

\[ f(x) = \begin{cases} 0 & -l \leq x < 0 \\ \sin \frac{\pi x}{l} & 0 \leq x < l \end{cases} \quad (13.54) \]

in a Fourier series. In accordance with Eq. (13.51), the coefficients would then be given by the integrals

\[ a_0 = \frac{1}{l} \int_0^l \sin \frac{\pi x}{l} dx \quad ; \quad a_n = \frac{1}{l} \int_0^l \sin \frac{\pi x}{l} \cos \frac{n\pi x}{l} dx \quad ; \quad b_n = \frac{1}{l} \int_0^l \sin \frac{\pi x}{l} \sin \frac{n\pi x}{l} dx \quad (13.55) \]

where $n = 1, 2, 3, \ldots$. We find these coefficients quickly by submitting to MAPLE the statements

> interface( showassumed=0 );
> assume( n, integer ) :
> a0 := int( sin(Pi*x/l), x=0..1 )/l;
\[ a0 := \frac{2}{\pi} \]
> an := int( sin(Pi*x/l)*cos(n*Pi*x/l), x=0..1 )/l;
\[ an := -\frac{1 + (-1)^n}{\pi (-1 + n^2)} \]
> a1 := int( sin(Pi*x/l)*cos(Pi*x/l),
x=0..l )/l;
a1 := 0

> bn := int( sin(Pi*x/l)*sin(n*Pi*x/l),
x=0..l )/l;
b1 := 1

> b1 := int( sin(Pi*x/l)*sin(Pi*x/l),
x=0..l )/l;
b1 := 1/2

Note expression for \(a_1\) is indeterminate, and find alternative evaluation.

Find \(b_n\), \(n \neq 1\).

Find \(b_1\), \(n = 1\).

Here, the second statement limits the nature of \(n\) so that MAPLE can automatically replace expressions like \(\sin n\pi\) and \(\cos n\pi\) with explicit evaluations, and the remaining statements evaluate the desired integrals. Thus, we find that

\[
a_0 = \frac{2}{\pi}; \quad a_1 = 0; \quad a_n = -\frac{1+(-1)^n}{\pi(n^2-1)}, n > 1; \quad b_1 = \frac{1}{2}; \quad b_n = 0, n > 1
\]  

(13.56)

Note, incidentally, that MAPLE gave us no warning that the value returned for \(b_n\) was, in fact, incorrect for \(n = 1\). We recognized its incorrectness by noting that, for \(n = 1\), the integrand is \(\sin^2(\pi x/l)\)—a quantity that is always positive, so its integral couldn’t possibly be zero.

With these values for the Fourier coefficients, we assemble the Fourier series in accordance with Eq. (13.50) to find that

\[
f(x) = \frac{1}{\pi} - \frac{2}{\pi} \sum_{n=2}^{\infty} \frac{1+(-1)^n}{\pi(n^2-1)} \cos \frac{n\pi x}{l} + \frac{1}{2} \sin \frac{\pi x}{l}
\]

\[
= \frac{1}{\pi} + \frac{1}{2} \sin \frac{\pi x}{l} - \frac{2}{\pi} \sum_{m=1}^{\infty} \frac{1}{4m^2-1} \cos \frac{2m\pi x}{l}
\]

\[
= \frac{1}{\pi} + \frac{1}{2} \sin \frac{\pi x}{l} - \frac{2}{\pi} \left( \frac{1}{3} \cos \frac{2\pi x}{l} + \frac{1}{15} \cos \frac{4\pi x}{l} + \frac{1}{35} \cos \frac{6\pi x}{l} + \cdots \right)
\]  

(13.57)

In the second and third forms, we have recognized that \(1+(-1)^n\) is zero when \(n\) is odd and 2 when \(n\) is even, so the sum on \(n\) can be extended over only even values of \(n\). Thus, setting \(n = 2m\), we have arranged in the second form for \(m\) to range over all positive integers.

The nature of this series is more clearly evident in a succession of graphs displaying the truncated series

\[
f_n(x) = \frac{1}{\pi} + \frac{1}{2} \sin \frac{\pi x}{l} - \frac{2}{\pi} \sum_{m=1}^{n} \frac{1}{4m^2-1} \cos \frac{2m\pi x}{l}
\]  

(13.58)

which approaches \(f(x)\) as \(n \to \infty\). Introducing the variable \(\overline{x} = x/l\) (and then dropping the overbar), we then generate graphs for various values of \(n\) with the statements listed in Ta ble 13.2. Here, we force \(m\) to be an integer, recast the general coefficient in an by replacing \(n\) with \(2m\), construct a MAPLE function defining the truncated series, and then plot the truncated series for each of four different truncation points. Shown in Fig. 13.6, the resulting graphs clearly reveal the convergence of the sum to the half-rectified wave as more terms are included.

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 formulae—for numerical integration. In this section, we describe several of these formulae.

\footnote{Note the use of the inert summation command \texttt{Sum}. Had we used the active command \texttt{sum}, MAPLE would have attempted to find a closed-form expression for the sum.}
13.5. ALGORITHMS FOR NUMERICAL INTEGRATION

Table 13.2: MAPLE code for graphing several Fourier series.

```maple
> assume( m, integer );
> simplify( subs( n=2*m, an ) );
> a := m -> -2/(Pi*(-1+4*m^2) );
> f := (x,n) -> 1/Pi + sin(Pi*x)/2 + Sum( a(m)*cos(2*m*Pi*x), m=1..n ):
> plot( f(x,1), x=-3.0..3.0, y=-0.5..1.5, color=black,
> thickness=3, labels=["X","F(X,1)"],
> labelfont=[TIMES,ROMAN,16], axesfont=[TIMES,ROMAN,14] );
> plot( f(x,2), x=-3.0..3.0, y=-0.5..1.5, color=black,
> thickness=3, labels=["X","F(X,2)"],
> labelfont=[TIMES,ROMAN,16], axesfont=[TIMES,ROMAN,14] );
> plot( f(x,4), x=-3.0..3.0, y=-0.5..1.5, color=black,
> thickness=3, labels=["X","F(X,4)"],
> labelfont=[TIMES,ROMAN,16], axesfont=[TIMES,ROMAN,14] );
> plot( f(x,8), x=-3.0..3.0, y=-0.5..1.5, color=black,
> thickness=3, labels=["X","F(X,8)"],
> labelfont=[TIMES,ROMAN,16], axesfont=[TIMES,ROMAN,14] );
> restart:
```

13.5.1 Newton-Cotes Quadrature

One family of quadrature formulae can be deduced by starting with the recognition that the definite integral

\[
A = \int_a^b f(x) \, dx
\]  

represents geometrically the area under the graph of \( f(x) \) over the interval \( a \leq x \leq b \). As shown in Fig. 13.7, let that interval be divided into \( N \) segments, each of width \( \Delta x = (b-a)/N \), let

\[
x_0 = a, \quad x_1 = a + \Delta x, \quad x_2 = a + 2 \Delta x, \ldots, \quad x_i = a + i \Delta x, \ldots, \quad x_N = b,
\]

and let \( f(x_0) = f(a) = f_0, \quad f(x_1) = f_1, \ldots, \quad f(x_i) = f_i, \ldots, \quad f(x_N) = f(b) = f_N \). 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. 13.8(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

\[
\int_a^b f(x) \, dx \approx M_N = \left( f_{1/2} + f_{3/2} + f_{5/2} + \cdots + f_{N-3/2} + f_{N-1/2} \right) \Delta x
\]
Figure 13.6: Succession of truncated series representing half-rectified sine wave. The upper left, upper right, lower left, and lower right graphs show the series when truncated at \( n = 1, 2, 4, \) and 8, respectively. These graphs were created with MAPLE.

Figure 13.7: Division of interval \( a < x < b \) into \( N \) segments.

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. 13.8(b)], in which case we obtain the trapezoidal rule,

\[
\int_a^b f(x) \, dx \approx T_N = \frac{1}{2} (f_0 + f_1) \Delta x + \frac{1}{2} (f_1 + f_2) \Delta x + \cdots + \frac{1}{2} (f_{N-1} + f_N) \Delta x
\]
Figure 13.8: Two different approximations leading to quadrature formulae. The left figure corresponds to Eq. (13.60), the right figure to Eq. (13.62).

\[ \int_a^b f(x) \, dx \approx S_N = \frac{1}{3} \left( f_0 + 4f_1 + 2f_2 + \cdots + 4f_{N-2} + 2f_{N-1} + f_N \right) \Delta x \] (13.63)

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,

\[ \frac{1}{3} \left( f_0 + 4f_1 + 2f_2 + \cdots + 4f_{N-2} + 2f_{N-1} + f_N \right) \Delta x \] (13.63)

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

\[ T_1 = T_{2^1} = \left( \frac{f(a) + f(b)}{2} \right) \frac{(b - a)}{2} \] (13.64)

Then, halving the step size and introducing \( x_1 = a + (b - a)/2 \), the midpoint of the interval (see Fig. 13.9, in which—for present convenience—we label the points differently than we did in Fig. 13.7), we find that

\[ T_2 = T_{2^1} = \left( \frac{f(a) + f(x_1) + f(b)}{2} \right) \frac{(b - a)}{2} = \frac{1}{2} T_1 + f(x_1) \frac{(b - a)}{2} \] (13.65)
Halving the step size again and introducing \( x_2 = a + (b - a)/4 \) and \( x_3 = a + 3(b - a)/4 \) (again, see Fig. 13.9), we find that

\[
T_4 = T_{2^2} = \left[ f(a) + f(x_2) + f(x_1) + f(x_3) + \frac{f(b)}{2} \right] \frac{(b-a)}{4}
\]

\[
= \frac{1}{2} T_2 + \left[ f(x_2) + f(x_3) \right] \frac{(b-a)}{4}
\]

\[
= \frac{1}{2} T_2 + \left[ f \left( a + \frac{b-a}{4} \right) + f \left( a + 3 \frac{b-a}{4} \right) \right] \frac{(b-a)}{4}
\]

(13.66)

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 continuing one more step, we find that

\[
T_8 = T_{2^3} = \frac{1}{2} T_4 + \left[ f(x_4) + f(x_5) + f(x_6) + f(x_7) \right] \frac{(b-a)}{8}
\]

\[
= \frac{1}{2} T_4 + \left[ f \left( a + \frac{b-a}{8} \right) + f \left( a + 3 \frac{b-a}{8} \right) + f \left( a + 5 \frac{b-a}{8} \right) \right]
\]

\[
+ f \left( a + 7 \frac{b-a}{8} \right) \frac{(b-a)}{8}
\]

(13.68)

and, in general, with

\[
x_{2^N+j} = a + \frac{(2j+1)(b-a)}{2^{N+1}} \quad ; \quad 0 \leq j < 2^{N+1}
\]

(13.69)

that

\[
T_{2^{N+1}} = \frac{1}{2} T_{2^N} + \left[ f \left( a + \frac{b-a}{2^{N+1}} \right) + f \left( a + 3 \frac{b-a}{2^{N+1}} \right) + \ldots \right.
\]

\[
+ f \left( a + (2^N - 1) \frac{b-a}{2^{N+1}} \right) \frac{(b-a)}{2^{N+1}}
\]

(13.70)

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. (13.62) and labeling the interpolation points as in Fig. 13.9, we find, for example, that

\[
T_4 = \left[ \frac{1}{2} f(a) + f(x_2) + f(x_1) + f(x_3) + \frac{1}{2} f(b) \right] \frac{b-a}{4}
\]

\[
= \left[ f(a) + 2f(x_2) + 2f(x_1) + 2f(x_3) + f(b) \right] \frac{b-a}{8}
\]

(13.71)
and that
\[
T_8 = \left[ \frac{1}{2} f(a) + f(x_4) + f(x_2) + f(x_5) + f(x_1) + f(x_6) + f(x_3) + f(x_7) + \frac{1}{2} f(b) \right] \frac{b-a}{8} \tag{13.72}
\]

Note, in particular, the combination
\[
\frac{4T_8 - T_4}{3} = \left[ f(a) + 4f(x_4) + 2f(x_2) + 4f(x_5) + 2f(x_1) + 4f(x_6) + 2f(x_3) + 4f(x_7) + f(b) \right] \left( \frac{b-a}{3 \times 8} \right) \tag{13.73}
\]

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
\[
\frac{4T_8 - T_4}{3} = \left[ f(a) + 4f(x_4) + 2f(x_2) + 4f(x_5) + 2f(x_1) + 4f(x_6) + 2f(x_3) + 4f(x_7) + f(b) \right] \left( \frac{\Delta x}{3} \right) \tag{13.74}
\]

which—*mirabile dictu*—we recognize as Simpson’s rule for evaluating the integral with 8 divisions of the interval! We conclude that
\[
S_8 = \frac{4T_8 - T_4}{3} \tag{13.75}
\]

More generally, we could also conclude that
\[
S_{2n} = \frac{4T_{2n} - T_n}{3} \tag{13.76}
\]

The extrapolation formula of Eq. (13.76) 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. (13.76), 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 \(T_n, T_{2n}, T_{4n}, \text{ and } T_{8n}\). We could then use Eq. (13.76) to generate the *three* values \(S_{2n}, S_{4n}, \text{ and } S_{8n}\). In Romberg integration, we next generate a pair of still more accurate values from the formulae
\[
X_{4n} = (16S_{4n} - S_{2n})/15 \quad \text{and} \quad X_{8n} = (16S_{8n} - S_{4n})/15;
\]

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 imprecision with which each component is represented within the computer
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 $10^3$, 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 strip width half of that used to determine the second value. 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).

From a more sophisticated perspective, numerical analysts have deduced expressions for the error in various Newton-Cotes formulae. For the midpoint formula of Eq. (13.61), for example,

$$
\left| \int_a^b f(x) \, dx - M_N \right| = \frac{(b - a)^3}{24N^2} \left| \frac{d^2 f}{dx^2} \right|_{x=\xi}
$$

(13.77)

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

$$
\left| \int_a^b f(x) \, dx - T_N \right| = \frac{(b - a)^3}{12N^2} \left| \frac{d^2 f}{dx^2} \right|_{x=\xi}
$$

(13.78)

and

$$
\left| \int_a^b f(x) \, dx - S_N \right| = \frac{(b - a)^5}{180N^4} \left| \frac{d^4 f}{dx^4} \right|_{x=\xi}
$$

(13.79)

can be derived for the trapezoidal rule given by Eq. (13.62) and for Simpson’s rule given by Eq. (13.63), 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 $d^4 f/dx^4 = 0$ 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.

---

8 We shall make this criterion a bit more explicit in the next paragraphs.
9 We shall see in later sections how we might decide when roundoff has started to be significant.
2. If \( D_{\text{max}}(i) \) is the maximum value of \( d^i f/dx^i \) in the interval \( a < x < b \), then the above expressions support the inequalities

\[
\left| \int_a^b f(x) \, dx - M_N \right| \leq \frac{(b-a)^3}{24N^2} D_{\text{max}}(2) \tag{13.80}
\]

\[
\left| \int_a^b f(x) \, dx - T_N \right| \leq \frac{(b-a)^3}{12N^2} D_{\text{max}}(2) \tag{13.81}
\]

\[
\left| \int_a^b f(x) \, dx - S_N \right| \leq \frac{(b-a)^5}{180N^4} D_{\text{max}}(4) \tag{13.82}
\]

(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 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 \( S_2, S_4, S_8, \ldots \), 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

\[
x = \frac{2t - (t_{i+1} + t_i)}{t_{i+1} - t_i} \quad \text{or} \quad t = \frac{t_{i+1} + t_i}{2} + \frac{t_{i+1} - t_i}{2} x = t_i^{\text{mid}} + \frac{\Delta t_i}{2} x
\]

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

\[
\int_a^b g(t) \, dt = \sum_{i=1}^m \int_{t_i}^{t_{i+1}} g(t) \, dt = \sum_{i=1}^m \frac{\Delta t_i}{2} \int_{-1}^{1} g\left( t_i^{\text{mid}} + \frac{\Delta t_i}{2} x \right) \, dx = \sum_{i=1}^m \frac{\Delta t_i}{2} \int_{-1}^{1} f_i(x) \, dx
\]

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

\[
\int_{-1}^{1} f(x) \, dx
\]

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

\[
\int_{-1}^{1} f(x) \, dx = \sum_{k=1}^N w_k f(x_k)
\]

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. (13.87) would assume the more explicit form

\[
\int_{-1}^{1} f(x) \, dx = w_1 f(x_1) + w_2 f(x_2)
\]

We choose \( w_1, \ w_2, \ x_1 \) and \( x_2 \) so that the formula gives the correct answer for the special cases

\[
f(x) = 1, \ \text{yielding that} \quad \int_{-1}^{1} dx = 2 = w_1 + w_2 \quad (13.89)
\]

\[
f(x) = x, \ \text{yielding that} \quad \int_{-1}^{1} x \, dx = 0 = w_1 x_1 + w_2 x_2 \quad (13.90)
\]

\[
f(x) = x^2, \ \text{yielding that} \quad \int_{-1}^{1} x^2 \, dx = \frac{2}{3} = w_1 x_1^2 + w_2 x_2^2 \quad (13.91)
\]

\[
f(x) = x^3, \ \text{yielding that} \quad \int_{-1}^{1} x^3 \, dx = 0 = w_1 x_1^3 + w_2 x_2^3 \quad (13.92)
\]
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. (13.90) and (13.92) imply that
\[ w_1 x_1 = -w_2 x_2 \quad \text{and} \quad w_1 x_1^3 = -w_2 x_2^3 \] (13.93)
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. (13.91) and (13.89) yield that \[ x_2 = \frac{1}{3} \] (13.94)
from which we also conclude that \[ x_1 = -\frac{1}{\sqrt{3}} \quad \text{and} \quad x_2 = \frac{1}{\sqrt{3}} \] (13.95)
Next, Eq. (13.90) implies that \[ w_1 = w_2 \] and then Eq. (13.89) implies that \[ w_1 = w_2 = 1. \] We conclude that
\[ \int_{-1}^{1} f(x) \, dx = f(-\frac{1}{\sqrt{3}}) + f\left(\frac{1}{\sqrt{3}}\right) \] (13.96)
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.\(^{11}\)
Returning to the original function and variables as laid out in Eq. (13.85), we finally find that Eq. (13.96) supports the expression
\[ \int_{a}^{b} g(t) \, dt = \sum_{i=1}^{m} \frac{\Delta t_i}{2} \left[ g\left(t_{\mid i} - \Delta t_i \frac{1}{\sqrt{3}}\right) + g\left(t_{\mid i} + \Delta t_i \frac{1}{\sqrt{3}}\right)\right] \]
\[ = \sum_{i=1}^{m} \frac{\Delta t_i}{2} \sum_{j=1}^{2} w_j g\left(t_{\mid i} + \Delta t_i \frac{1}{\sqrt{3}} x_j\right) \] (13.97)
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
\[ \int_{-1}^{1} f(x) \, dx = 0.23692689 f(-0.90617985) + 0.47862867 f(-0.53846931) \]
\[ + 0.56888889 f(0.00000000) \]
\[ + 0.47862867 f(0.53846931) + 0.23692689 f(0.90617985) \] (13.98)
which is 100% accurate for polynomials of the 9th-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
\[ w_1 = 0.23692689 \quad x_1 = -0.90617985 \]
\[ w_2 = 0.47862867 \quad x_2 = -0.53846931 \]
\[ w_3 = 0.56888889 \quad x_3 = +0.00000000 \]\n\[ w_4 = 0.47862867 \quad x_4 = +0.53846931 \]
\[ w_5 = 0.23692689 \quad x_5 = +0.90617985 \] (13.99)
\(^{11}\)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. (13.87) increases.
Further, returning to the original variable, we note that
\[
\int_{a}^{b} g(t) \, dt = \sum_{i=1}^{m} \frac{\Delta t_i}{2} \sum_{j=1}^{5} w_j \, g \left( t_{mid}^{i} + \frac{\Delta t_i}{2} \, x_j \right) \tag{13.100}
\]

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 \( (2/((1 - x_i^2))(dL_2(x_i)/dx)^2) \). More generally, for an \( N \)-point Gaussian integration, we would discover that\(^{12}\)
\[
L_N(x_i) = 0 \quad \text{and} \quad w_i = \frac{2}{(1 - x_i^2)(dL_N(x_i)/dx)^2} \tag{13.101}
\]
and that
\[
\left| \int_{-1}^{1} f(x) \, dx - \sum_{k=1}^{N} w_k \, f(x_k) \right| = \frac{2^{2N+1}(N!)^4}{(2N + 1)(2N)!^3} \left| \frac{d^{2N} f}{dx^{2N}} \right|_{x=\xi} \tag{13.102}
\]
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 Gauss-Legendre formulae.\(^{13}\)

### 13.6 Evaluating Integrals Numerically with IDL

**Note:** All IDL program (.pro) files referred to in this chapter are available in the directory \$HEAD/idl, where (as defined in the Local Guide) \$HEAD must be replaced by the appropriate path for your site. At some sites, this directory or some other directory containing these files may also have been placed in IDL’s default search path. If so, the files can be found by IDL without explicit specification of a path. Otherwise, you will have to use the full path to copy them into your default directory to access them.

One dimensional integrals can be evaluated numerically either by using IDL’s elementary commands or, more simply, by invoking one of the built-in IDL routines \( qsimp \) and \( qromb \) or the Lawrence-provided routine \( luqsimp \). We mention that IDL also provides the routines \( \text{int}_2d \) and \( \text{int}_3d \) for evaluating two- and three-dimensional integrals and the routine \( \text{int}_\text{tabulated} \) for integrating functions whose values are supplied in a table (as might arise with experimental data). We shall here illustrate only \( luqsimp \), but we invite you to study the IDL manuals for information on the other available routines.

#### 13.6.1 Using Elementary Commands

Relatively simple sequences of elementary commands can implement one or another of the algorithms described in Section 13.5. If, for example, we seek an evaluation of the integral
\[
I = \text{erf}(1) = \frac{2}{\sqrt{\pi}} \int_{0}^{1} e^{-x^2} \, dx \tag{13.103}
\]
\(^{13}\)A more general integral that can be approached with the techniques of this subsection has the form \( \int_{b}^{a} f(x) \, w(x) \, dx \), 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 \( L_i(x) \) 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.
Table 13.3: Values of erf(1.0) obtained by the trapezoidal rule and a user-constructed program in IDL.

<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>32</td>
<td>0.842633</td>
<td>1024</td>
<td>0.842701</td>
</tr>
<tr>
<td>2</td>
<td>0.825263</td>
<td>64</td>
<td>0.842684</td>
<td>2048</td>
<td>0.842700</td>
</tr>
<tr>
<td>4</td>
<td>0.838368</td>
<td>128</td>
<td>0.842696</td>
<td>4096</td>
<td>0.842701</td>
</tr>
<tr>
<td>8</td>
<td>0.841619</td>
<td>256</td>
<td>0.842700</td>
<td>8192</td>
<td>0.842701</td>
</tr>
<tr>
<td>16</td>
<td>0.842430</td>
<td>512</td>
<td>0.842700</td>
<td>16384</td>
<td>0.842701</td>
</tr>
</tbody>
</table>

which appears in the expression of Eq. (13.41) for the probability of finding a quantum oscillator in the classically forbidden region, we might invoke the trapezoidal rule as presented in Eq. (13.62), divide the interval $0 \leq x \leq 1$ into $n = 4$ segments and use the statements:\footnote{Alternatively, as long as $n \geq 2$, the fifth, sixth, and seventh statements in this code can be replaced with the single statement $y = 0.5*f[0] + \text{total}(f[1:n+1]) + 0.5*f[n]$—a statement that will execute more quickly.}

\begin{verbatim}
IDL> n = 4
IDL> x = findgen(n+1)/float(n)
IDL> dx = 1.0/float(n)
IDL> f = 2.0*exp(-x^2)/sqrt(!pi)
IDL> y = 0.5*f[0]
IDL> for i = 1,n-1 do y=y+f[i]
IDL> y = y + 0.5*f[n]
IDL> print, dx*y

0.838368
\end{verbatim}

To illustrate the reduction of truncation error with decreasing step size, we re-execute this procedure, starting with $n = 1$ and successively doubling $n$, finding the values shown in Table 13.3. (The process comes to an end at $n = 16384$ because, for $n = 32768$, the particular coding we have adopted requires us to have more than 32767 ($= 2^{16} - 1$) elements in the vectors $x$ and $f$ and the 16-bit integer used for the index in those arrays cannot store positive integers larger than 32767.) On the basis of this succession of numbers, however, we conclude that, to six decimal places, the value of the integral is somewhere between 0.842700 and 0.842701.\footnote{The fluctuations in the last digit are almost certainly a reflection of internal computer roundoff, which—in single precision arithmetic—can easily affect the sixth or seventh “significant” digit.} The trapezoidal rule took something like 257 or 513 evaluations of the function to reach that value.

Note in this example that the result for $n = 1$ is quite inaccurate. As $n$ increases, however, the result becomes more and more accurate. After a time (here at about $n = 256$) the result stabilizes and further increase of $n$ makes almost no difference in the value. A graph of $T_n$ versus $n$ would start low, increase to this stable value and remain horizontal as $n$ is further increased. Indeed, in this example, that graph would remain horizontal all the way to $n = 16384$. Were we able to increase $n$ still further, we would sooner or later find that this graph would begin to depart from the value at which it stabilized, at first slowly but then more dramatically. The point at which that departure begins is the point at which roundoff errors begin to become significant. Fortunately, $n$ exhibits a substantial range of values that are simultaneously large enough to keep truncation error at bay and small enough to prevent significant roundoff error. Short of changing to a more sophisticated algorithm or of shifting, for example, from single to double precision arithmetic, the most accurate value we can obtain is the value at which further increase of $n$ (for awhile) makes little difference in the value obtained.
We illustrate with three alternative and more efficient approaches. First, note that Richardson extrapolation applied to $T_1$ and $T_2$ and then again to $T_2$ and $T_4$ yields the values

$$S_2 = \frac{4 \cdot T_2 - T_1}{3} = \frac{4 \cdot 0.825263 - 0.771743}{3} = 0.843103 \quad (13.104)$$

and

$$S_4 = \frac{4 \cdot T_4 - T_2}{3} = \frac{4 \cdot 0.838368 - 0.825263}{3} = 0.842736 \quad (13.105)$$

Continuing this process of Richardson extrapolation, we find the values $S_8 = 0.842700$, $S_{32} = 0.842701$, and $S_{64} = 0.842701$. Because $S_{32}$ and $S_{64}$ agree to six digits, we conclude that $I = 0.842701$ to the sixth decimal place. We have here arrived at the same conclusion as in the previous paragraph, but only 33 evaluations of the function were necessary. The trapezoidal rule coupled with Richardson extrapolation is clearly more efficient than the trapezoidal rule alone, at least with this integral.

Second, if we instead adopt Simpson’s rule as in Eq. (13.63), we would modify the above procedure to recognize that each evaluation of the function requires a different weight, i.e., the first and last values of the function must be multiplied by 1, the second value and alternate values thereafter must be multiplied by 4, and the third value and alternate values thereafter must be multiplied by 2. To achieve this end, we create a vector of weights having the value $[1,4,2,4,\ldots,2,4,1]$. We might invoke the statements

```idl
IDL> n = 2
IDL> x = findgen(n+1)/float(n)
IDL> dx = 1.0/float(n)
IDL> f = 2.0*exp(-x^2)/sqrt(!pi)
IDL> w = fltarr(n+1) + 2.0
IDL> w[0] = 1.0 & w[n] = 1.0
IDL> for i=1,n-1,2 do w[i]=4.0
IDL> y = 0.0
IDL> for i = 0,n do y=y+w[i]*f[i]
IDL> print, dx*y/3.0
0.843103
```

If this process is repeated with $n = 4$, the end result is 0.842736. With $n = 32$ and $n = 64$, this process yields the values 0.842700 and 0.842701, respectively. Reassuringly, all of these values agree with the values obtained by applying Richardson extrapolation to the values obtained in the previous paragraph with the trapezoidal rule. We again conclude that, to six digits, $I = 0.842701$.

Third, we could adopt the Gaussian approach. If, for example, we identify $t$ in Eq. (13.97) with $x$ in Eq. (13.103) and $g(t)$ with $2e^{-x^2}/\sqrt{\pi}$ and we divide the interval $0 \leq x \leq 1$ into $m = 4$ segments of equal width, we might invoke the IDL statements

```idl
IDL> pts = [-1.0/sqrt(3.0), 1/sqrt(3.0)]
IDL> w = [1.0, 1.0]
IDL> a = 0.0 & b = 1.0
```

To set the evaluation points, weights, and limits. Then we would execute the statements

```idl
IDL> m = 4
IDL> dx = (b-a)/float(m)
IDL> x = dx * findgen(m)+a
IDL> xmid = fltarr(m)
IDL> for i=0,m-1 do xmid[i]=(x[i]+x[i+1])/2.0
```

16Alternatively, the sum of the products of the components of $w$ and $f$ could be viewed as a multicomponent vector dot product and evaluated with the single statement $y = w \# transpose(f)$.

17Remember that indices in IDL start at 0. Thus, values pertaining to the first segment will be identified by index 0, those pertaining to the second segment by 1, \ldots, and those pertaining to the $m$-th segment by $m-1$. 

to set several parameters and determine the coordinates at the midpoints of the segments. Finally, we evaluate the sum, multiply with an overall factor whose inclusion was postponed, and display the result with the statements

```
IDL> int = 0.0
IDL> for i=0,m-1 do begin $
IDL> fcts = 0.0 & $
IDL> for j = 0,1 do fcts = fcts + w[j]*exp(-(xmid[i]+dx*pts[j]/2.0)^2) & $
IDL> int = int + fcts & endfor
IDL> int = dx*int/sqrt(!pi)
IDL> print,int
 0.842699
```

finding that the two-point Gaussian formula with four equal divisions of the interval of integration yields a result that is correct to five digits. With 1, 2, 4, and 8 divisions, the results are 0.842442, 0.842677, 0.842699, and 0.842701, respectively—and we find that the two-point Gaussian formula yields a result correct to four digits even with only two divisions of the interval of integration and correct to six digits with eight divisions.

The coding worked out in the previous paragraph is easily adapted to express the five-point Gaussian formula. We begin by invoking the statements

```
IDL> pts = [ -0.90617985, -0.53846931, 0.0, 0.53846931, 0.90617985 ]
IDL> w = [ 0.23692689, 0.47862867, 0.56888889, 0.47862867, 0.23692689 ]
IDL> a = 0.0 & b = 1.0
```

to set the evaluation points $x_1, \ldots, x_5$, the weights $w_1, \ldots, w_5$, and the limits. Then, we would execute the statements

```
IDL> m = 4
IDL> dx = (b-a)/float(m)
IDL> x = dx * findgen(m+1) + a
IDL> xmid = fltarr(m)
IDL> for i=0,m-1 do xmid[i]=(x[i]+x[i+1])/2.0
IDL> int = 0.0
IDL> for i=0,m-1 do begin $
IDL> fcts = 0.0 & $
IDL> for j = 0,4 do fcts = fcts + w[j]*exp(-(xmid[i]+dx*pts[j]/2.0)^2) & $
IDL> int = int + fcts & endfor
IDL> int = dx*int/sqrt(!pi)
IDL> print,int
 0.842701
```

finding that the five-point Gaussian formula with four equal divisions of the interval of integration yields a result that is correct to six digits. Even more amazing, this five-point formula gives the same result—0.842701—with $m = 1$, i.e., when the entire interval of integration is treated as a single segment!
Table 13.4: The IDL function \texttt{gausint.pro}.

\begin{verbatim}
function gausint, t
;
; GAUSINT defines the integrand for evaluating the error function.
;
tmp1 = 2.0/sqrt( !Pi )
return, tmp1*exp(-t^2)
end
\end{verbatim}

13.6.2 The Function \texttt{luqsimp}

For our purposes, the most suitable IDL routine for the numerical evaluation of one-dimensional integrals is a Lawrence-written adaptation \texttt{luqsimp} of the routine \texttt{qsimpson} written and contributed to the IDL Astronomy User's Library by W. Landsman.\footnote{The routine \texttt{luqsimp.pro} and the routine \texttt{trapzd.pro}, which—renamed \texttt{lutrapzd.pro} in the present context—is called by \texttt{luqsimp.pro}, are used here and distributed with this book by permission of W. Landsman and of Research Systems, Inc. (RSI). Dr. Landsman wrote the original code, and employees of RSI contributed a small modification of Landsman's \texttt{qsimpson.pro} before the Lawrence modifications were added. Dr. Landsman's source code can be retrieved from the IDL Astronomy User's Library by accessing the URL \url{http://idlastro.gsfc.nasa.gov/homepage.html}. Full contact information for RSI can be found in Section Z.} It is based on the routine \texttt{qsimp} described in \textit{Numerical Recipes} by William H. Press, \textit{et al.} (Cambridge University Press, Cambridge, MA, 1988) and is written in the IDL language. The \texttt{1u} modification is cast as a function to facilitate its use with all three types of integral as identified at the very beginning of this chapter. The routine \texttt{luqsimp} uses both of the strategies described in Section 13.5.2 to achieve efficiencies in execution. Before invoking \texttt{luqsimp}, the user must create a function pro-file defining the integrand as a function of a single scalar argument (the integration variable). The construction of such files is described in Section 2.8. For the integral in Eq. (13.41), for example, the function pro-file might be as shown in Table 13.4.

Having stored the pro-file in the default directory with a name identical to that of the function name and with the file extension \texttt{.pro} (here \texttt{gausint.pro}), we would evaluate the integral and display its value with the statements

\begin{verbatim}
q = luqsimp( 'function_name', a, b, eps=value, max_iter=value, tol=value );
print, q
\end{verbatim}

where \texttt{function_name}, which is mandatory, identifies the file in which the integrand is defined; \texttt{a} and \texttt{b}—also mandatory—are the lower and upper limits of integration; the optional keyword \texttt{eps} (default $10^{-6}$) specifies the desired \textit{fractional} accuracy; the optional keyword \texttt{max_iter} (default 20) determines the maximum number of evaluations of the function by the formula $2^{\text{max}_\text{iter}}$; and the optional keyword \texttt{tol} (default 0.0) specifies the \textit{absolute} accuracy. The routine \texttt{luqsimp} returns the value of the integral in the specified variable, here \texttt{q}. Finally, the \texttt{print} procedure displays the value returned by \texttt{luqsimp}.

For the example of Eq. (13.41), the integral is evaluated—one with the fractional accuracy set to $10^{-3}$ and again with the default fractional accuracy ($10^{-6}$)—and displayed by the statements
function moment, lambda
    ; MOMENT defines the integrand for evaluating the moment of
    ; inertia of a semicircular plate.
    ;
    tmp = lambda^2
    return, 4.0*tmp*sqrt(1.0-tmp)/!pi
end

IDL> q = luqsimp( 'gausint', 0.0, 1.0, eps = 0.001 )
IDL> print, q
  0.84273603
IDL> q = luqsimp( 'gausint', 0.0, 1.0 )
IDL> print, q
  0.84270076

Since these values agree to four digits, we conclude that both (0.8427) are accurate to four digits. The second is surely a good deal more accurate, but we would have to rely on attainment of the default value \( \text{eps} = 10^{-6} \) before asserting—correctly—that the value of the integral is 0.842701 to six digits.

IDL also has several built-in routines for evaluating one-dimensional integrals, including \texttt{qsimp} and \texttt{qromb}, which use Simpson's rule and Romberg integration, respectively, and are based on similarly named functions described in \textit{Numerical Recipes} by William H. Press, \textit{et al.} (Cambridge University Press, Cambridge, MA, 1988). Because these routines do not afford the option of specifying the error as an absolute error, they will sometimes have difficulties with integrals whose value is, in fact, zero. For that reason, we have elected to focus this section on \texttt{luqsimp}. The reader is referred to the IDL manuals for information about \texttt{qsimp} and \texttt{qromb}.

13.6.3 Moment of Inertia

To evaluate the integral appearing in Eq. (13.16) for the moment of inertia of a semicircular plate, we must first create the function pro-file listed in Table 13.5 and store it in a file named \texttt{moment.pro}.

Then, the simple statements

IDL> I = luqsimp( 'moment', 0.0, 1.0, eps=0.001 )
IDL> print, I
  0.24989874
IDL> I = luqsimp( 'moment', 0.0, 1.0 )
IDL> print, I
  0.24999992

evaluate the integral to one part in \(10^3\) and \(10^6\), respectively, and display the results. These results are clearly in agreement with one another and with those obtained by other methods in this chapter.

13.6.4 Quantum Probabilities

In Section 13.6.2, we have already evaluated the integral that appears in the determination of the probability that a quantum harmonic oscillator in its ground state will be found outside the classical
turning point. According to Eq. (13.41), that probability is given by the expression

\[ P(|x| > |x_{\text{turn}}|) = 1.0 - \text{erf}(1.0) = 1.0 - 0.842701 = 0.157299 \] (13.106)

We conclude that, in a bit over 15% of the measurements, the quantum oscillator will be found in the classically forbidden region!

### 13.6.5 Integrals as Functions of the Upper Limit

To evaluate an integral as a function of a parameter, `luqsimp` must be invoked repeatedly in a loop. With each execution of the loop, the parameter assumes a new value and the current values of the parameter and of the corresponding integral are stored for later examination. This process is easiest to implement when the parameter is the upper limit of the integral as, for example, in the integral

\[ g(x) = \int_{0}^{x} f(t) \, dt \] (13.107)

We begin by constructing a function pro-file to define the integrand \( f(t) \), say `integ.pro`. Then we invoke the IDL statements

```idl
x = findgen ...  
q = size(x) & N = q[1] - 1  
g = fltarr(N+1)  
for i = 0, N do 
  g[i] = luqsimp( 'integ', 0, x[i] )
```

Upon execution of these statements, values of \( g(x) \) as a function of \( x \) for the selected values of \( x \) will be stored in the vector \( g \) and can be further processed as desired. In particular, the statement `plot,x,g` will generate a graph of \( g \) versus \( x \).

### 13.6.6 The Error Function

To evaluate the error function as given by Eq. (13.26), we begin by constructing the pro-file giving the integrand. The function `gausint.pro` defined in Section 13.6.2 can be used here also. Thus, supposing we want \( x \) to range from 0.0 to 3.0 in steps of 0.1, we invoke the IDL statements

```idl
IDL> x = findgen(31)/10.0  
IDL> q = size(x) & N = q[1] - 1  
IDL> erf = fltarr(N+1)  
IDL> for i = 0, N do 
  erf[i] = luqsimp( 'gausint',0,x[i] )  
IDL> plot, x,erf, title = 'Error Function', thick=3.0, charsize=1.5, xtitle= 'x', ytitle='erf(x)', ticklen=1.0
```

The resulting graph is shown in Fig. 13.10.

---

19This sequence of statements actually is computationally inefficient. We might increase the efficiency by recognizing, for example, that \( g(x + \Delta x) = g(x) + \int_{x}^{x+\Delta x} f(t) \, dt \) and obtain integrals for larger \( x \) by adding an appropriate increment to already evaluated integrals for smaller \( x \). For our present purposes, that approach unnecessarily complicates the algorithm of evaluation.

20The warning message declaring that the “sum did not converge after 20 steps” can be ignored. It comes because the very first integral of the 101 integrals evaluated has the value zero, and a convergence criterion based on fractional error is not compatible with an integral whose value is zero. The message can be suppressed by specifying an absolute tolerance, setting `eps=0.0` and `tol = 0.00001`, say. That step is, however, not necessary because, error message to the contrary notwithstanding, the evaluation has, in fact, converged.
13.6. The Cornu Spiral

To determine the Cornu spiral, we use \texttt{luqsimp} to generate vectors \( C \) and \( S \) containing values of the defining integrals as given in Eq. (13.27) over a suitable range of upper limits and then plot the values in \( S \) versus the values in \( C \). First we create the two pro-files listed in Table 13.6 to provide the two integrands. Rather than storing these definitions in the user’s default directory in the separate files \texttt{spiralc.pro} and \texttt{spirals.pro}, we elect to place both definitions in the single file \texttt{cornu.pro}.\footnote{With this structure, however, automatic compilation will not occur when each function is first named, so the statement \texttt{.compile cornu} must be explicitly executed before either function can be invoked.}

For definiteness, we elect to explore the spiral over the range \(-5 \leq u \leq 5\), though we evaluate only

\begin{verbatim}
function spiralc, t
    ; SPIRALC defines the integrand for the integral giving the
    ; vertical coordinate of the Cornu spiral.
    ;
    return, cos( !pi * t^2 / 2 )
end

function spirals, t
    ; SPIRALS defines the integrand for the integral giving the
    ; horizontal coordinate of the Cornu spiral.
    ;
    return, sin( !pi * t^2 / 2 )
end
\end{verbatim}
integrals for $0 \leq u \leq 5$ and obtain values for $-5 \leq u \leq 0$ by recognizing that both integrals are odd functions of $u$. First, we compile the functions `spiralc` and `spirals` and prepare necessary arrays with the statements

```idl
IDL> .compile cornu
IDL> u = findgen(101)/20.0
IDL> q = size(u) & N = q[1] - 1
IDL> CP = fltarr(N+1)
IDL> SP = CP & CN = CP & SN = CP
```

Compile needed functions.
Specify values of upper limit.
Set $N$ to highest index in $u$.
Establish arrays for integrals.

Then we evaluate the integrals with the single (composite) statement

```idl
IDL> for i = 0, N do begin $  
IDL> CP[i]=luqsimp( 'spiralc', 0, u[i] ) & $  
IDL> SP[i]=luqsimp( 'spirals', 0, u[i] ) & $  
IDL> CN[N - i] = -CP[i] & $  
IDL> SN[N - i] = -SP[i] & $  
IDL> end
```

Evaluate/store integrals for each limit.
Create vectors for negative limits.

Finally, we assemble the variables to be plotted and plot the graph with the statements

```idl
IDL> C = [CN, CP]
IDL> S = [SN, SP]
IDL> window, 1, xsize=400, ysize=430
IDL> !x.margin = [ 8.0, 2.0 ]
IDL> !y.margin = [ 5.0, 5.0 ]
IDL> plot, C, S, title='Cornu Spiral', ticklen=1.0, charsize=1.25, $  
IDL> xtitle='C(u)', ytitle='S(u)', thick=3.0
```

The resulting graph is shown in Fig. 13.11.\(^{22,23}\)

\(^{22}\)This sequence of statements actually has two glitches. First, it is computationally inefficient in the way described in footnote 19. Second, and less significantly, the values of $S(0)$ and $C(0)$ appear twice in the vectors finally plotted.

\(^{23}\)See Section 2.16.5 for a description of the strategy invoked to produce equal scaling on both axes.
Table 13.7: Structure of IDL function file with parameter in the integrand.

```idl
function function_name, x
    ; Explanatory comments.
    ;
    common param, p1, p2, p3

... Statements to evaluate integrand, using p1, p2, and p3 for the parameters.
return, integrand
end
```

13.6.8 Integrals as Functions of an Internal Parameter

The situation in which an integral of interest is a function of a parameter in the integrand is more difficult because the integration routine `luqsimp` does not have an argument for specifying a value for a parameter to be passed to the user-defined function pro-file that returns values of the integrand to `luqsimp`. Instead, we make use of named common,\(^{24}\) which is a programming construct built to facilitate assigning values to variables in one routine and passing those values to another routine without the values necessarily being known to any intervening routines. Thus, we can use named common to give a value to a parameter at the main command level and pass that value to the function defining the integrand without passing it through `luqsimp`. To invoke this feature, we must do two things:

1. We add one line to the function pro-file that defines the integrand, so that the file has the format shown in Table 13.7. Here, common is a mandatory IDL reserved word, param is a user-assigned name for the common area, and p1, p2, and p3—however many are needed—are the variables that will store the parameters. The quantities function name, x, and integrand have the same meaning as before, and the file must be stored with the name function_name.pro.

2. Having defined the function pro-file in this new way, we then invoke IDL to evaluate and display the desired integral by using `luqsimp` with the statements

   ```idl
   common param, p1, p2, p3
   p1 = ?? & p2 = ?? & p3 = ??
   q = luqsimp( 'function_name', a, b )
   print, q
   ```

   At command level, the common statement need be invoked only once, regardless of how many different values of the parameters are to be explored. Further, variables referenced in the common statement cannot previously have been defined.\(^{25}\)

Note that the names of the parameters need not be the same at command level and in the function pro-file. Only the position and data type of the entities must match in the two occurrences, since it is the order and type of items—not the variable names used—that provide the association of values in the two occurrences.

13.6.9 The Off-Axis Electrostatic Potential of Two Rings

Equation (13.36) illustrates a situation in which the integral is a function of an internal parameter. For the integral in Eq. (13.36), for example, we might write the pro-file listed in Table 13.8. We

\(^{24}\)This construction was introduced in Section 2.8, which see.

\(^{25}\)Remember the IDL command `delvar` for removing variables that are no longer needed.
Table 13.8: The IDL function rings.pro.

```idl
function rings, phi

; RINGS defines the integrand whose integral gives the
; electrostatic potential in the mid-plane between two
; uniformly charged circular rings.

; common param, s
tmp = sqrt( 2.0 ) / ( 2.0 * !Pi )
y = tmp / sqrt( 2.0 - 2.0 * s * cos(phi) + s * s )
return, y
end
```

store this file in the default directory with the name rings.pro, then invoke IDL to evaluate and display the desired integral by using luqsimp with the statements

```idl
common param, s
s = (suitable value)
q = luqsimp( 'rings', a, b )
print, q
```

Since we seek to explore this integral as a function of s over, say, 0.0 ≤ s ≤ 4.0, we must generate a vector of values of s and then, in a loop, invoke luqsimp once for each element in that vector. We might use the statements

```idl
IDL> common param, ss
IDL> s = findgen(41)/10.0
IDL> q = size(s) & N = q[1] - 1
IDL> V = fltarr(N+1)
IDL> for i = 0, N do begin
   ss = s[i] & V[i] = luqsimp( 'rings', 0.0, 2.0*!pi ) & end
IDL> plot, s, V, title = 'Potential in Midplane', thick=3.0, $
IDL>   xtitle = 's (in units of a)', charsize=1.25, $
IDL>   ytitle = 'V(s) (in units of V(0))', ticklen=1.0
```

The resulting graph is shown in Fig. 13.12.

### 13.11 Evaluating Integrals Numerically with MAPLE

To evaluate an integral numerically using MAPLE, we simply embed the command int or the command Int within the command evalf. In response to the simple command

```maple
evalf( int( f, x=a..b ) );
```

where \( f \) is an expression that depends on \( x \), MAPLE first attempts to evaluate \( \int_a^b f \, dx \) symbolically and then evaluates the numerical value of the symbolic result. If, however, the attempt at symbolic integration fails, MAPLE first submits the integral to some built-in routines based on algorithms from the Numerical Algorithms Group (NAG) library and, if that fails, adopts by default the Clenshaw-Curtis quadrature algorithm and includes some features for dealing with singularities of the integrand.
detected in (or near) the interval of integration—to produce the numerical result. The alternative form

\[ \text{evalf( Int( f, x=a..b );) ;} \]

causes MAPLE to skip any attempt at symbolic integration and go directly to evaluation by a NAG routine and then, if necessary, by Clenshaw-Curtis quadrature. The more elaborate form

\[ \text{evalf( Int( f, x=a..b, digits=dig, method=flag );) ;} \]

provides control over the tolerance and the method. Here,

- \( \text{dig} \) specifies the number of digits of accuracy in the returned value, where the default value of \( \text{dig} \) is taken to be the value of the environment variable \text{Digits}, typically 10.

- \( \text{flag} \) specifies the method, and may have one of the values \_CCquad (Clenshaw-Curtis quadrature but with no attempt to detect or deal with singularities), \_Dexp (adaptive double-exponential routine), and \_NCrule (Newton-Cotes quadrature using quanc8, an adaptive eighth-order method).

Fuller detail about these features of MAPLE can be found in the MAPLE manuals.

With this feature of MAPLE, we can evaluate our standard example in many ways. We might, for example, use the statement

\[ > \text{evalf( int( 2*exp(-t^2)/sqrt(Pi), t=0..1 );)} ; \]

\[ .8427007929 \]

which will evaluate the integral analytically \[\text{obtaining erf(1)}\] and then evaluate the error function of argument 1. Alternatively, we might use the statement

\[ > \text{evalf( Int( 2*exp(-t^2)/sqrt(Pi), t=0..1 );) ;} \]

\[ .8427007929 \]
> evalf( Int( 2*exp(-t^2)/sqrt(Pi), t=0..1 ) );
.8427007929

which jumps directly to numerical integration. We might reduce the accuracy—and speed the
evaluation—with the result with the statement

> evalf( Int( 2*exp(-t^2)/sqrt(Pi), t=0..1, digits=3 ) );
.843

or preserve the default accuracy but use the Newton-Cotes method with the statement

> evalf( Int( 2.0*exp(-t^2)/sqrt(Pi), t=0.0..1.0, method=_NCrule ) );
.8427007929

Reassuringly, the results are all consistent with one another.

13.11.1 Quantum Probability

In the opening paragraphs of this section, we have already evaluated the integral that appears in
the determination of the probability that a quantum harmonic oscillator in its ground state will be
found outside the classical turning point. According to Eq. (13.41), that probability is given by the
expression

\[
P(|x| > |x_{\text{turn}}|) = 1.0 - \text{erf}(1.0) = 1.0 - 0.8427007929 = .1572992071
\]  

(13.108)

which is (probably) correct to ten digits after the decimal point. We conclude that in a bit over 15%
of the measurements, the quantum oscillator will be found in the classically forbidden region!

13.11.2 The Error Function

To define the function \texttt{gaussian} and then generate successive numerical values of the function \texttt{erf(x)}
defined by Eq. (13.26) as \(x\) varies from 0.0 to 3.0 in steps of 0.1, we might use the statements

\begin{verbatim}
> q := Array( 1..31 ):
> gaussian := t -> 2*exp(-t^2)/sqrt(Pi):
> for j from 1 to 31 do
>     q[j] := [ evalf( (j-1)/10 ), evalf(Int(gaussian(x), x=0..(j-1)/10)) ]:
> end do:
\end{verbatim}

The \texttt{Array} \texttt{q} of two-component lists created by these statements must then be converted into a \texttt{list}

\begin{verbatim}
> q1 := convert( q, list ):
\end{verbatim}

before the graph of Fig. 13.13 can be created with the statement

\begin{verbatim}
> plots[listplot]( q1, color='black', labels=['x','Error Function'], thickness=3, labelfont=[TIMES,ROMAN,16], axesfont=[TIMES,ROMAN,14], labeldirections=[HORIZONTAL,VERTICAL] );
\end{verbatim}
13.11.3 The Off-Axis Electrostatic Potential of Two Rings

Using MAPLE, we could evaluate and plot the integral in Eq. (13.36) to find the electrostatic potential at the radial coordinate $s_a$ in the midplane between two identical, uniformly charged parallel rings of radius $a$. The statement

```maple
> rings := (phi,s) -> sqrt(2)/(2*Pi*sqrt(2-2*s*cos(phi)+s^2)):
```

defines the integrand, and the statements

```maple
> v := Array(1..31):
> for ss from 1 to 31 do
> s := (ss-1)/10:
> v[ss] := [evalf(s), evalf(Int(rings(phi,s), phi=0..2*Pi, method=_NCrule))]:
> end do:
> v1 := convert(v, list):
```

establish a suitable array, evaluate the integral as a function of $s$ for $0.0 \leq s \leq 3.0$ in steps of 0.1, and create the appropriate list of lists to facilitate plotting. Finally, the statement

```maple
> plots[listplot](v1, color='black', labels=['s','Potential'], thickness=3, labelfont=[TIMES,ROMAN,16], axesfont=[TIMES,ROMAN,14], labelfont=[HORIZONTAL,VERTICAL], view=[0..3,0..1]);
```

produces the graph shown in Fig. 13.14.
13.13 Evaluating Integrals Numerically with FORTRAN

Note: Except for files explicitly flagged as from the Numerical Recipes library, all FORTRAN programs (*.f) and all FORTRAN-created data files (*.f.dat) in this chapter can be copied from the directory $HEAD/fortran, where (as defined in the Local Guide) $HEAD must be replaced by the appropriate path for your site.

FORTRAN 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 Numerical Recipes library—and devote our efforts solely to the easier task of writing a suitable driving program to invoke the features of whatever integration subroutine we choose to use.

13.13.1 Writing Programs from Scratch

More to illustrate the essence of an algorithm than to develop a truly useful integration program, we begin by presenting a quick FORTRAN program to implement the trapezoidal rule as expressed in Eq. (13.62). The program contains three major sections. The first section obtains controlling input (integration limits and desired number of segments) and is expressed with the coding

\[ \text{\textcopyright 2023, The McGraw-Hill Companies, Inc. All rights reserved.} \]
The second section evaluates the integral with the coding

\[
\text{DX} = \frac{\text{B-A}}{\text{F}	ext{LOAT(\text{N})}} \quad ! \text{Set size of segment} \\
\text{VALUE} = 0.5 \times \text{FUNC(A)} \quad ! \text{Compute first term} \\
\text{DO I = 1, N-1} \\
\quad \text{VALUE} = \text{VALUE} + \text{FUNC( A + I*DX )} \\
\text{ENDDO} \\
\text{VALUE} = \text{VALUE} + 0.5 \times \text{FUNC(B)} \quad ! \text{Add last term} \\
\text{VALUE} = \text{VALUE} \times \text{DX} \quad ! \text{Compute integral}
\]

Finally, the third section displays the resulting evaluation with the coding

\[
\text{WRITE(*, '(1X,A,F10.6)') 'Integral = ', VALUE}
\]

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

\[
\begin{align*}
\text{FUNCTION} & \quad \text{FUNC}(\text{X}) \\
\text{FUNC} & = 2.0 \times \text{EXP}(-\text{X**2})/\text{SQRT}(3.14159265)
\end{align*}
\]

which can be placed either before or after the program itself. A full listing of this program, named trapezoidal.f, is presented in Section 13.A. 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.f 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), we execute the statements:\footnote{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. 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 \textit{Local Guide}.}

\[
f77 -o \text{trapezoidal.xf} \text{trapezoidal.f} \quad \text{Create executable file.} \\
./\text{trapezoidal.xf} \quad \text{Execute program.} \\
\text{Lower limit : 0.0} \quad \text{Enter controlling values.} \\
\text{Upper limit : 1.0} \\
\text{Number of segments: 1} \\
\text{Integral = 0.771743} \quad \text{View output.}
\]

The results for various divisions of the integration interval are shown in Table 13.9. Note in particular that, as \(n\) increases from 1, the results change noticeably (reflecting \textit{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 \textit{increasing roundoff} error). This table provides evidence that there indeed is a fairly wide range in which \(n\) is simultaneously large enough (\(\Delta x\) is small enough) to give accurate results but not so large (\(\Delta x\) not so small) as to trigger problems from computer roundoff. We conclude that, to six digits, the value of this integral is 0.842701.
Table 13.9: Values of $\text{erf}(1.0)$ obtained with the trapezoidal rule and the FORTRAN program trapezoidal.f.

<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>0.842726</td>
</tr>
<tr>
<td>16</td>
<td>0.842430</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>0.843682</td>
</tr>
<tr>
<td>64</td>
<td>0.842684</td>
<td>16384</td>
<td>0.842701</td>
<td>4194304</td>
<td>0.839947</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>

This example, of course, illustrates only how to write a FORTRAN 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 named common and edit the function defining the integrand to refer to that named common for the value of the parameter.\(^{29}\)

### 13.13.2 Using Numerical Recipes

*Numerical Recipes* provides many subroutines to integrate functions numerically, including `trapzd.f`, `qtrap.f`, `qsimp.f`, and `qromb.f`. Since it can be used by itself and it is also used in the interior of the other routines, `trapzd.f` is the workhouse for the entire suite. It is called with the statement

```fortran
CALL TRAPZD( FUNC, A, B, S, I )
```

When invoked, `trapzd.f` evaluates the integral of the function defined by `func` over the limits from `A` to `B`, dividing that interval into $2^{I-1}$ segments\(^{30}\) and returning the result in the variable `S`. Note, however, that this subroutine 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, `trapzd.f` evaluates $T_1$ and then—recursively—$T_2$, $T_4$, $T_8$, ..., using the efficiency described in the first paragraph of Section 13.5.2.

A FORTRAN driving program for using `trapzd.f` must be written by the user and will have the general form in Table 13.10.\(^{31}\) This program defines the function to be integrated, establishes the desired limits, invokes `trapzd` in a properly constructed loop, and displays each value as it is obtained.

---

\(^{29}\)Details on the use of named common are discussed in Section 9.5.4; further illustration will be found in Section 11.14.

\(^{30}\)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.

\(^{31}\)A somewhat more elaborate demonstration program named `xtrapzd.f` can be found in the directory `$NRHEAD/recipes_f/demo/src`.
For the sake of an explicit example, we have chosen to evaluate 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 xtrapzd.f from the directory $NRHEAD/recipes_f/demo/src to the default directory, giving it the name errtrapzd.f and editing it appropriately, or by creating the file errtrapzd.f from scratch with an available text editor.

2. Copy the recipe trapzd.f from the directory $NRHEAD/recipes_f/recipes to the default directory.

3. Compile and link the executable module with the statement

   ```
f77 -o errtrapzd.xf errtrapzd.f trapzd.f
   ```

4. Run the program with the statement ./errtrapzd.xf.

The output from the program will be that shown in Table 13.11. 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* subroutine qtrap.f uses subroutine trapzd.f in a loop that evaluates the integral repeatedly, assessing convergence at each step, and stops either when two consecutive values differ by less than \( 10^{-4}\% \) of the first of the two [as defined by the editable parameter eps (default \( 10^{-6} \)) in the subroutine qtrap.f] or when more than 20 refinements of the step size [as defined by the editable parameter jmax (default 20)] have been effected without convergence.\(^{32}\) It is invoked with a statement of the form

\(^{32}\)In some cases, e.g., when the value of the integral being addressed is actually zero, it may be prudent to edit qtrap.f to replace the convergence criterion based on fractional change with one based on absolute change in consecutive iterates.

---

Table 13.10: The FORTRAN program ERRTRAPZD.F.

```FORTRAN

PROGRAM ERRTRAPZD
EXTERNAL FUNC ! Alert compiler that FUNC is a user-defined
! procedure that will appear as an argument
! to a subroutine
NUMIT = 14 ! Set number of iterations
A = 0.0 ! Set lower limit
B = 1.0 ! Set upper limit

! Head table, then invoke TRAPZD iteratively
! and display result at each step
WRITE(*,'(1X,A)') ' n No Divs Approx. Integral'
DO I = 1, NUMIT
   CALL TRAPZD( FUNC, A, B, S, I )
   WRITE(*,'(1X,I6,I10, F20.6)') I, 2**(I-1), S
ENDDO
END

FUNCTION FUNC(X) ! Define integrand
   FUNC= 2.0*EXP(-X**2)/SQRT(3.14159265)
END
```

---

For the sake of an explicit example, we have chosen to evaluate erf(1.0) = \( \frac{2}{\sqrt{\pi}} \int_0^1 e^{-x^2} \, dx \).
Table 13.11: Output from FORTRAN program ERRTRAPZD.F.

<table>
<thead>
<tr>
<th>n</th>
<th>No Divs</th>
<th>Approx. Integral</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0.771743</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0.825263</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>0.838368</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>0.841619</td>
</tr>
<tr>
<td>5</td>
<td>16</td>
<td>0.842430</td>
</tr>
<tr>
<td>6</td>
<td>32</td>
<td>0.842633</td>
</tr>
<tr>
<td>7</td>
<td>64</td>
<td>0.842684</td>
</tr>
<tr>
<td>8</td>
<td>128</td>
<td>0.842697</td>
</tr>
<tr>
<td>9</td>
<td>256</td>
<td>0.842700</td>
</tr>
<tr>
<td>10</td>
<td>512</td>
<td>0.842701</td>
</tr>
<tr>
<td>11</td>
<td>1024</td>
<td>0.842701</td>
</tr>
<tr>
<td>12</td>
<td>2048</td>
<td>0.842701</td>
</tr>
<tr>
<td>13</td>
<td>4096</td>
<td>0.842701</td>
</tr>
<tr>
<td>14</td>
<td>8192</td>
<td>0.842701</td>
</tr>
</tbody>
</table>

Table 13.12: The FORTRAN program ERRQTRAP.F.

```fortran
PROGRAM ERRQTRAP
  EXTERNAL FUNC          ! Admit that function may be defined elsewhere
  A = 0.0              ! Set lower limit
  B = 1.0              ! Set upper limit

  ! Head table, then invoke QTRAP and
  ! display result
  CALL QTRAP( FUNC, A, B, S )
  WRITE(*, '(1X,A,F10.6)') 'Result using QTRAP = ', S
END

FUNCTION FUNC(X)        ! Define integrand
  FUNC= 2.0*EXP(-X**2)/SQRT(3.14159265)
END
```

CALL QTRAP( FUNC, A, B, S )

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.f is simpler than one using trapzd.f directly and might have the form listed in Table 13.12. The steps in evaluating the desired integral with this program are

1. Create the driving program, either by copying the demonstration program xqtrap.f from the directory $NRHEAD/recipes.f/demo/src to the default directory, giving it the name errqtrap.f, and editing it appropriately, or by creating the file errqtrap.f from scratch with an available text editor.

2. Copy any necessary recipes to the default directory. In the present case, the needed recipe trapzd.f presumably remains from the first example. The recipe qtrap.f must also be copied
13.14. EVALUATING INTEGRALS NUMERICALLY WITH C

from the directory $NRHEAD/recipes_f/recipes.

3. Compile and link the executable module with the statement

    f77 -o errqtrap.xf errqtrap.f trapzd.f qtrap.f

4. Run the program with the statement ./errqtrap.xf

The output from the program will be

Result using QTRAP = 0.842701

which is certainly in agreement with the result to which errtrapzd.f converged. (Note that failure
to converge will result in a message “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.f. After printing this message, the program pauses. Proper response—see the rest
of the message—will result in continued execution with the assignment of the current value to the
integral (even though convergence was not attained); any other response will terminate execution of
the program.)

The Numerical Recipes subroutines qsimp.f and qromb.f are used in a way parallel to that of
qtrap.f. 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 named common and edit the function defining the integrand to refer to
that named common for the value of parameter.33

13.14 Evaluating Integrals Numerically with C

Note: Except for files explicitly flagged as from the Numerical Recipes library, all C
programs (*.c) and all C-created data files (*.c.dat) in this chapter can be copied
from the directory $HEAD/cc, where (as defined in the Local Guide) $HEAD must be
replaced by the appropriate path for your site.

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
Numerical Recipes library—and devote our efforts solely to the easier task of writing a suitable
driving program to invoke the features of whatever integration subroutine we choose.

33See footnote 29.
13.14.1 Writing Programs from Scratch

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. (13.62). 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 */
```

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 );
value = value + 0.5*func(b); /* Add last term */
value = value * dx; /* Compute integral */
```

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

34To 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.

35Don’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 13.13: Values of erf(1.0) obtained with the trapezoidal rule and the C program `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>0.842726</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>0.843682</td>
</tr>
<tr>
<td>64</td>
<td>0.842684</td>
<td>16384</td>
<td>0.842701</td>
<td>4194304</td>
<td>0.839947</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>

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 13.13. 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 enough ($\Delta x$ is small enough) to give accurate results but not so large ($\Delta x$ 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 and then using the declared variable name both in the function and in the main program when values are assigned to the parameter).^36

### 13.14.2 Using Numerical Recipes

*Numerical Recipes* provides many integration procedures cast in the form of functions, including `trapzd.c`, `qtrap.c`, `qsimp.c`, and `qromb.c`. Since it can be used by itself and it is also used in the interior of the other routines, `trapzd.c` is the workhouse of the entire suite. It is called by the statement

```c
s = trapzd( func, a, b, i );
```

^36^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.
TABLE 13.14: THE C PROGRAM ERRTRAPZD.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", "n", "approx. integral");
  for (i=1;i<=numit;i++)
  {
    s=trapzd( func, a, b, i );
    printf("%6d %20.6f\n", i, s);
  }
}

When invoked, trapzd.c evaluates the integral of the function defined by func over the limits from a to b, dividing that interval into \(2^{i-1}\) segments\(^{37}\) 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, 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 trapzd.c must be written by the user and will have the general form listed in Table 13.14.\(^{38}\) This program defines the function to be integrated, establishes the desired limits, invokes 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 \(\text{erf}(1.0) = \frac{2}{\sqrt{2\pi}} \int_0^1 e^{-x^2} \, dx\).

The steps in evaluating the desired integral with this program then are

1. Create the driving program, either by copying the demonstration program xtrapzd.c from the

---

\(^{37}\)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.

\(^{38}\)A somewhat more elaborate demonstration program named xtrapzd.c can be found in the directory \$NRHEAD/recipes_c-ansi/recipes\ or—possibly—in the directory \$NRHEAD/recipes_c-kr/recipes. (See the Local Guide.)
Table 13.15: 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>

directory\(^{*}\) `$/NRHEAD/recipes_c-ansi/demo/src` to the default directory, giving it the name `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 13.15. 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.\(^{40}\) It is invoked with a statement of the form

```
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 13.16. The steps in evaluating the desired integral with this program are

\(^{39}\)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 `$/NRHEAD/recipes_c-kr/...`

\(^{40}\)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.
CHAPTER 13. EVALUATING INTEGRALS

Table 13.16: 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 */
}
```

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).

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 $v(0) = v_0$.

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 $x(0) = x_0$ and $v(0) = v_0$.

13.4. Suppose an object of mass $m$ moves non-relativistically in one dimension under the action of the force $f(t) = f_0e^{-bt}$, 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 $\bar{x}$ and variance $\sigma^2$, defined by

$$\bar{x} = \lim_{b \to \infty} \int_{-b}^{+b} x p(x) \, dx \quad \text{and} \quad \sigma^2 = \lim_{b \to \infty} \int_{-b}^{+b} (x - \bar{x})^2 p(x) \, dx$$

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

$$\frac{1}{2} m \left(\frac{dx}{dt}\right)^2 - G\frac{mM}{x} = -G\frac{mM}{x_0} \quad \Rightarrow \quad \frac{dx}{dt} = \sqrt{2GM} \sqrt{\frac{1}{x} - \frac{1}{x_0}}$$

41See footnote 36.
384

CHAPTER 13. EVALUATING INTEGRALS

Consider a surface in the

13.9. In a spherically symmetric charge distribution, the charge density is a function of only the radial

coordinate,

\[ \frac{\rho}{r} \]

given in polar coordinates by the function

\[ r(a, \theta, \phi) = a(1 - \cos \phi) \]

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.

\[ T_{\text{midhalf}} = \frac{1}{\sqrt{2GM}} \int_{x_0/4}^{3x_0/4} \left( \frac{1}{x} - \frac{1}{x_0} \right)^{-1/2} \, dx \]

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

\[ P(r) = \frac{4}{a^3} \int_0^r e^{-2\rho^2} r'^2 \, dr' = 4 \int_0^{r/a} e^{-2\rho^2} \rho^2 \, d\rho \]

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 \((0,0,a)\) 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

\[ V(z) = \frac{\sigma}{2\pi\epsilon_0} \left[ \sqrt{a^2 + (z-ca)^2} - |z-ca| - \sqrt{a^2 + (z+ca)^2} + |z+ca| \right] \]

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, \( r = z\hat{k} \), for which \( z > a \) by setting up and evaluating the integral

\[ V(0,0,z) = \frac{1}{4\pi\epsilon_0} \int \frac{\rho(r') \, dv'}{|r - r'|} \]

symbolically to show that \( V(0,0,z) = Q/4\pi\epsilon_0z \). \( (Q \) 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 \( r' = x' \hat{i} + y' \hat{j} + z' \hat{k} \) 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
\[ x_{mn} = \frac{16a}{\pi^2} (-1)^{(m+n+1)/2} \frac{mn}{(m^2 - n^2)^2} \]
\[ p_{mn} = \frac{2i\hbar}{a} (-1)^{(m+n+1)/2} \frac{mn}{(m^2 - n^2)} \]
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 \(N = 0, 1, 2, 3, 4, 5, \text{and } 6\).

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

\[
L_0(x) = 1 \\
L_1(x) = x \\
L_2(x) = \frac{1}{4}(3x^2 - 1) \\
L_3(x) = \frac{1}{2}(5x^3 - 3x) \\
L_4(x) = \frac{1}{8}(35x^4 - 30x^2 + 3) \\
L_5(x) = \frac{1}{16}(63x^5 - 70x^3 + 15x) \\
L_6(x) = \frac{1}{16}(231x^6 - 315x^4 + 105x^2 - 5) \\
L_7(x) = \frac{1}{35}(429x^7 - 693x^5 + 315x^3 - 35x) \\
L_8(x) = \frac{1}{128}(6435x^8 - 12012x^6 + 6930x^4 - 1260x^2 + 35) \\
L_9(x) = \frac{1}{1536}(12155x^9 - 25740x^7 + 18018x^5 - 4620x^3 + 315x)
\]

### 13.15.2 . . . using Numerical Methods

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. (13.63). 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.

#### 13.15.
Redo several of the examples in Sections 13.6.3–13.6.9 using the built-in IDL routines \texttt{qsimp} and/or \texttt{qromb}. Be sure to include at least one example in each of the three categories (integral that is a single number, integral as a function of a parameter in the limits, integral as a function of a parameter in the integrand). Write a paragraph describing any differences you observe among the several available routines.

#### 13.17. (a) Deduce the three-point Gaussian formula, for which—paralleling Eq. (13.88)—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. (13.101) where \( L_4(x) = (5x^4 - 3x)/2. \)

#### 13.18.
The (normalized) wave functions for a quantum harmonic oscillator in its first and second excited states \( (n = 1, n = 2) \) are

\[
\psi_1(x) = \sqrt{\frac{1}{2}} \left( \frac{m\omega}{\pi\hbar} \right)^{1/4} y e^{-y^2/2} ; \quad \psi_2(x) = \frac{1}{\sqrt{2}} \left( \frac{m\omega}{\pi\hbar} \right)^{1/4} (2y^2 - 1) e^{-y^2/2}
\]

where \( y = x/\sqrt{\hbar \omega/k}, \) the energies of these states are \( 3\hbar \omega/2, \) and \( 5\hbar \omega/2, \) 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

\[
f(v) = 4\pi \left( \frac{m}{2\pi kT} \right)^{1/2} \int_0^v e^{-mv^2/2kT} v' \, dv'
\]

for the fraction of the molecules having speed less than \( v. \) 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 \( 4 \times 10^{14} \text{ Hz} \leq \nu \leq 7 \times 10^{14} \text{ Hz} \) as a function of temperature. Hint: One way to approach this exercise would be to choose a reference frequency \( \nu_0 \) arbitrarily (say \( 10^{14} \text{ Hz} \)) and recast the integral on the dimensionless variable \( s = \nu/\nu_0 \). Examination of \( I \) in units of \( 8\pi h\nu_0^4/c^3 \) 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

\[ P(a, b) = \frac{1}{\sqrt{2\pi}} \int_a^b e^{-t^2/(2\sigma^2)} dt \]

Thus, the probability of finding a value between \( a \) and \( b \) is given by

Show analytically that \( P(-x, x) = \text{erf}(x/(\sqrt{2}\sigma)) \), 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? Hint: Since the gravitational potential is \(-GmM/x\), conservation of energy yields

\[ \frac{1}{2} \frac{dx}{dt}^2 - G\frac{mM}{x} = -G\frac{mM}{x_0} \quad \Rightarrow \quad \frac{dx}{dt} = -\sqrt{2GM} \sqrt{\frac{1}{x} - \frac{1}{x_0}} \]

(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 \( 2\pi \sqrt{x_0^3/GM} \). For the earth around the sun, this latter time is, of course, 1 year. Optional: See if you can develop a means to determine the time required for the first half of the journey, which unfortunately—for numerical approaches—involves a convergent but improper integral.

13.23. The normalized Lorentz distribution function is given by

\[ p(x) = \frac{1}{\pi} \frac{a/2}{x^2 + (a/2)^2} \]

Find the probability that a single, randomly selected value will be in the range \(-a \leq x \leq a\). Make sure to assess the precision of your result by methods that do not exploit a priori knowledge of the exact value. Hint: Before evaluating the integral, introduce the dimensionless variable \( s = x/a \) and note that the result actually doesn’t depend on \( a \), so there is but one number to determine.
13.24. 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

\[
P(r) = \frac{A}{a^3} \int_0^r e^{-2r'/a} r'^2 \, dr' = 4 \int_0^{r/a} e^{-2\rho} \rho^2 \, d\rho
\]

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 \( r/a \).

13.25. The complete elliptic integrals of the first and second kinds are given by

\[
K(k) = \int_0^{\pi/2} \frac{d\phi}{(1 - k^2 \sin^2 \phi)^{1/2}} ; \quad E(k) = \int_0^{\pi/2} (1 - k^2 \sin^2 \phi)^{1/2} \, d\phi
\]

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.

13.26. 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 \( \beta = \sin^{-1}(|\sin(\theta/2)|/k) \). Obtain graphs of \( \theta(t) \) versus \( \omega t \) for several different values of \( k \).

13.27. The \( n \)-th order Bessel function can be defined by the integral

\[
J_n(x) = \frac{1}{\pi} \int_0^\pi \cos(nx - x \sin \theta) \, d\theta
\]

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

\[
V(r, \phi, z) = \frac{Q}{4\pi \varepsilon_0 a} = \frac{1}{\pi} \int_0^n \left(1 - 2 \frac{r}{a} \cos \phi' + \frac{r^2}{a^2} + \frac{z^2}{a^2}\right)^{-1/2} \, d\phi'
\]

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

\[
\mathbf{B}(x, z) = \frac{\mu_0 I'/2\pi a}{x^2 + z^2 + a^2 - 2ax \cos \phi'} \hat{\mathbf{z}}
\]

Explore both components of this magnetic field numerically as functions of \( x/a \) for various values of \( z/a \), including \( z/a = 0.0 \) (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.

13.32. Patterning your program after the program `trapezoidal.f`, write and test a FORTRAN program
that uses Simpson’s rule [Eq. (13.63)] to evaluate

\[ I = \frac{2}{\sqrt{\pi}} \int_a^b e^{-x^2} \, dx \]

13.33. Write and test a FORTRAN program that uses the five-point Gaussian formula to evaluate the
integral

\[ I = \frac{2}{\sqrt{\pi}} \int_a^b e^{-x^2} \, dx \]

13.34. Patterning your program after the program `trapezoidal.c`, write and test a C program that uses
Simpson’s rule [Eq. (13.63)] 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

\textit{Note:} Numerical recipes can, of course, also be used for any of the
exercises in Section 13.15.2.

13.36. Edit the program `errtrapzd.f` 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.

13.37. Following the pattern illustrated in Section 13.13.2, create and test FORTRAN programs
`errqsimp.f` and `errqromb.f` using `qsimp.f` and `qromb.f`, respectively, to evaluate \( \text{erf}(1.0) \). Note
that, in addition to `trapzd.f`, `qromb.f` also invokes `polint.f`, which will have to be available and
included in the compile instruction before `errqromb.f` will compile successfully.
13.38. Edit the driving program `errqtrap.f` (or—better—the driving program `errqsimp.f` 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.

13.39. Edit the driving program `errqtrap.f` (or—better—the driving program `errqsimp.f` 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 erf(\( x \)) as defined in Eq. (13.26) as a test integrand for your program.

13.40. Use `trapzd.f`, `qtrap.f`, `qsimp.f`, and `qromb.f` 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 FORTRAN are introduced in Chapter 9.

13.41. Use Numerical Recipes in FORTRAN to address Exercise 13.19, which involves evaluating an integral as a function of its upper limit.

13.42. Use Numerical Recipes in FORTRAN 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 COMMON storage in FORTRAN.

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 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.A  Listing of trapezoidal.f

! Program trapezoidal.f

! This FORTRAN 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.

! ***** DEFINE INTEGRAND AS FUNCTION *****

FUNCTION FUNC(X)
    FUNC = 2.0*EXP(-X**2)/SQRT(3.14159265)
END

PROGRAM TRAPEZOIDAL

! ***** READ LIMITS, NUMBER OF SEGMENTS, *****

WRITE(*, '(1X,A)') 'Lower limit : ',
READ(*,*) A
WRITE(*, '(1X,A)') 'Upper limit : ',
READ(*,*) B
WRITE(*, '(1X,A)') 'Number of segments: ',
READ(*,*) N

! ***** EVALUATE INTEGRAL *****

DX = (B-A)/FLOAT(N) ! Set size of segment
VALUE = 0.5 * FUNC(A) ! Compute first term
DO I = 1, N-1 ! Add middle terms
   VALUE = VALUE + FUNC( A + I*DX )
ENDDO
VALUE = VALUE + 0.5 * FUNC(B) ! Add last term
VALUE = VALUE * DX ! Compute integral

! ***** DISPLAY RESULT *****

WRITE(*, '(1X,A,F10.6)') 'Integral = ', VALUE

END
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.

¹The graphs could, of course, be produced in any number of ways. Except for Fig. 14.2 (which was produced with 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

\[
V(x) = E \quad \text{or} \quad V(x) - E = 0 \quad (14.2)
\]

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

\[
2 V(x) = x^3 + 10000 + x^2 - 200 - x - 1/2 \quad (14.3)
\]

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

\[
-50.00 < x_1 < -47.50 \quad ; \quad -12.50 < x_2 < -10.00 \quad ; \quad 7.50 < x_3 < 10.00 \quad (14.4)
\]

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

\[
F = 0 \quad \implies \quad \frac{dV}{dx} = 0 \quad (14.5)
\]

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

\[
\frac{dV}{dx} = \frac{3x^2}{10000} + \frac{x}{100} - \frac{1}{500} = 0 \quad (14.6)
\]

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

\[
-35.0 < x_4 < -30.0 \quad \text{and} \quad -2.5 < x_5 < 2.5 \quad (14.7)
\]

\[\text{This function is, of course, not particularly realistic as a potential energy. We can, however, provide a physical context for at least a portion of } V(x). \text{ If we confine our attention to the region around the one minimum it possesses, we can interpret the function as the potential energy of an anharmonic oscillator, for which—if that minimum occurs at } x = 0 \text{—we might write } V(x) = kx^2/2 + ax^4 \text{ while imposing the constraint that } a, \text{ the coefficient of the cubic perturbation from the potential energy of a simple harmonic oscillator, be small. The expression in Eq. (14.2) simply places the minimum at a different value of } x \text{ and adds a constant to the potential energy. In what follows, we will explore this function over a wider range of values of } x \text{ than is physically meaningful. The pedagogic advantage of Eq. (14.2) is that it combines many of the important features of potential energy functions with, as we shall see, tractability by a variety of different approaches.}\]
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\(^3\)

\[
\begin{align*}
    m \frac{d^2 x_1}{dt^2} &= -kx_1 + k'(x_2 - x_1) \\
    m \frac{d^2 x_2}{dt^2} &= -kx_2 - k'(x_2 - x_1)
\end{align*}
\] (14.8)

To cast these equations in dimensionless form, we choose a unit of length \( a \), set \( x_i/a = \bar{x}_i \), introduce \( \omega_0 = \sqrt{k/m} \), set \( \bar{k} = k'/k \), and introduce \( \bar{t} = \omega_0 t \) to find that

\[
\begin{align*}
    \frac{d^2 \bar{x}_1}{d\bar{t}^2} &= -\bar{k} \bar{x}_1 + \bar{k} \bar{x}_2 \\
    \frac{d^2 \bar{x}_2}{d\bar{t}^2} &= \bar{k} \bar{x}_1 - (1 + \bar{k}) \bar{x}_2
\end{align*}
\] (14.9)

Next, seeking sinusoidal (or simple harmonic) solutions, we suppose that

\[
\begin{align*}
    \bar{x}_1(\bar{t}) &= \bar{x}_{10} \cos \omega \bar{t} \\
    \bar{x}_2(\bar{t}) &= \bar{x}_{20} \cos \omega \bar{t}
\end{align*}
\] (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

\[
\begin{pmatrix}
1 + \kappa - \omega^2 & -\kappa \\
-\kappa & 1 + \kappa - \omega^2
\end{pmatrix}
\begin{pmatrix}
x_{10} \\
x_{20}
\end{pmatrix} = 0
\] (14.11)

The solution of this equation for the unknowns \( x_{10} \) and \( x_{20} \) 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

\[
D(\omega) = (1 + \kappa - \omega^2)^2 - \kappa^2 = \omega^4 - 2\omega^2(1 + \kappa) + 1 + 2\kappa = 0
\] (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

$$m \frac{d^2 x}{dt^2} = -b \frac{dx}{dt} \quad \text{and} \quad m \frac{d^2 y}{dt^2} = -mg - b \frac{dy}{dt}$$

(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

$$x(t) = m v_0 \cos \theta \frac{1 - e^{-bt/m}}{b}$$

(14.15)

$$y(t) = - \frac{mgt}{b} + m \frac{v_0 \sin \theta + \frac{mg}{b}}{b} \left(1 - e^{-bt/m}\right)$$

(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

$$- \frac{mgt}{b} + m \frac{v_0 \sin \theta + \frac{mg}{b}}{b} \left(1 - e^{-bt/m}\right) = 0$$

(14.17)

and then evaluate $x(t_1)$. Equivalently, if we introduce the dimensionless time $\tau = bt/m$ and the dimensionless parameter $\alpha = \frac{bv_0}{mg}$, the equation whose root we seek becomes

$$f(\tau) = \tau - (1 + \alpha \sin \theta) \left(1 - e^{-\tau}\right) = 0$$

(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

$$R(\theta, \alpha) = \frac{v_0}{g} \frac{\cos \theta (1 - e^{-\tau_1})}{\alpha} = \frac{\cos \theta}{\alpha (1 + \alpha \sin \theta)} \tau_1$$

(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 $\alpha = \frac{bv_0}{mg}$.
- 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).
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

\[0 = \tau - (1 + \alpha \sin \theta) \left(1 - \left(1 - \tau + \frac{\tau^2}{2}\right)\right) = \tau + (1 + \alpha \sin \theta) \left[-\tau + \frac{\tau^2}{2}\right]\]

\[= -\alpha \tau \sin \theta + \frac{\tau^2}{2} = 0 \implies \tau = 2\alpha \sin \theta \tag{14.20}\]

Then, substituting this result into Eq. (14.19), we find that

\[
\frac{R(\theta, \alpha \to 0)}{v_0^2/g} = \frac{\cos \theta}{\alpha(1 + \alpha \sin \theta)} (2\alpha \sin \theta) = \frac{2 \sin \theta \cos \theta}{(1 + \alpha \sin \theta)} \to 2 \sin \theta \cos \theta \tag{14.21}
\]

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

$$V(x) = \begin{cases} \infty & x < 0 \\ -V_0 & 0 \leq x \leq a \\ 0 & a \leq x \end{cases}$$  \hspace{1cm} (14.22)

leads to the conclusion that the allowed energies $E$ should satisfy the equation

$$s \cot s = -\sqrt{c^2 - s^2}$$  \hspace{1cm} (14.23)

where $c^2 = 2ma^2V_0/\hbar^2$ 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 $1 + \cot^2 \theta = \sin^{-2} \theta$ to find that

$$s^2 \cot^2 s = s^2 (\sin^{-2} s - 1) = c^2 - s^2 \implies \sin s = \pm \frac{s}{c}$$  \hspace{1cm} (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

$$\sin s = \pm 0.04s \quad \text{or} \quad \sin s \mp 0.04s = 0$$  \hspace{1cm} (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 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

$$\begin{align*}
\text{root 1u} & : -0.5 < s < 0.5 \\
\text{root 2u} & : 2.5 < s < 3.5 \\
\text{root 3u} & : 6.0 < s < 7.0 \\
\text{root 4u} & : 8.5 < s < 9.5 \\
\text{root 5u} & : 12.5 < s < 13.5 \\
\text{root 6u} & : 14.5 < s < 15.5 \\
\text{root 7u} & : 19.0 < s < 20.0 \\
\text{root 8u} & : 20.5 < s < 21.5 \\
\end{align*}$$  \hspace{1cm} (14.26)

and that, for the lower sign, the nine roots are bounded—again crudely—by

$$\begin{align*}
\text{root 1l} & : -0.5 < s < 0.5 \\
\text{root 2l} & : 3.0 < s < 4.0 \\
\text{root 3l} & : 5.5 < s < 6.5 \\
\text{root 4l} & : 9.5 < s < 10.5 \\
\text{root 5l} & : 11.5 < s < 12.5 \\
\text{root 6l} & : 16.0 < s < 17.0 \\
\text{root 7l} & : 17.5 < s < 18.5 \\
\text{root 8l} & : 22.5 < s < 23.5 \\
\text{root 9l} & : 23.5 < s < 24.5 \\
\end{align*}$$  \hspace{1cm} (14.27)

$^5$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.
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

\[ f(x) = ax + b = 0 \quad \Rightarrow \quad x = -\frac{b}{a} \quad (14.28) \]

and that the quadratic formula

\[ f(x) = ax^2 + bx + c = 0 \quad \Rightarrow \quad x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} = \frac{2c}{-b \pm \sqrt{b^2 - 4ac}} \quad (14.29) \]

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

\[ f(x) = \sin(kx) = 0 \quad \Rightarrow \quad x = \frac{n\pi}{k}, \quad n = 0, \pm 1, \pm 2, \ldots \quad (14.30) \]
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

\[ f(x) = \frac{a + x}{b + x^2} - c \]  

(14.31)

for example, can be found symbolically because, so long as \( b + x^2 \neq 0 \), the simple recasting

\[ \frac{a + x}{b + x^2} - c = 0 \implies a + x - c(b + x^2) = -cx^2 + x + a - cb = 0 \]  

(14.32)

converts the problem into one involving a quadratic polynomial. Beyond these few cases, very few roots can be found symbolically.

### 14.4 Finding Roots Symbolically with MAPLE

The MAPLE command for solving a wide variety of equations is `solve`. It takes two arguments, the first giving the equation to be solved (or the function whose roots are desired) and the second specifying the variable for which a solution is desired. Thus, the statement

\[
\texttt{solve( a*x + b = 0, x );}
\]

returns the solution \( x = -b/a \) and the statement

\[
\texttt{solve( a*x^2 + b*x + c , x );}
\]

\[
\frac{1}{2} \left( -b + \sqrt{b^2 - 4ac} \right) a, -\frac{1}{2} \left( b + \sqrt{b^2 - 4ac} \right) a
\]

in which MAPLE assumes the omitted `= 0` at the end of the first argument, returns the sequence

\[
\frac{1}{2} \left( -b + \sqrt{b^2 - 4ac} \right) a, \frac{1}{2} \left( b - \sqrt{b^2 - 4ac} \right) a
\]  

(14.33)

containing two solutions. The command `solve` actually also knows how to deal with cubic and quartic polynomials, though the result in the most general case is likely to be extremely involved and will be useful only if, once obtained, it is quickly converted to floating point form. The command can solve higher order polynomials only if it recognizes a factorization into polynomials, no one of which is of order higher than quartic.

Sometimes (e.g., for roots of polynomials of order greater than three), `solve` may be unable to find explicit solutions. In those cases, `solve` returns the root in the form `RootOf(...)`, e.g.,

\[
\texttt{solve(a*x^6-b*x^5 + c = 0, x);}
\]

\[
\text{RootOf}(a.Z^6 - b.Z^5 + c)
\]

(Here, \( Z \) is a MAPLE-generated generic variable.) Basically, this response constitutes an admission by MAPLE that it can’t solve the specified equation. Note also that the command `solve` is affected by several environment variables, including

- `_EnvExplicit`, which controls the extent to which irrational solutions will be returned by `solve`. By default, this variable is undefined, which results in the display of all rational solutions and some irrational solutions—those that aren’t too complicated. To display all irrational solutions (that MAPLE can find), we would set this variable to `true`; set to `false`, the variable causes all irrational solutions to be returned as `RootOfs`. 


• **EnvDropMultiplicity** which controls the multiple display of multiple roots. Set the variable to `true` to suppress repeated display of multiple roots.

• **MaxSols**, which specifies the number of solutions to be found. This variable is undefined by default.

• **EnvAllSolutions**, which we would set to `true` if we wanted MAPLE to try to find and display *all* solutions to the specified equation. This variable is undefined by default.

Fuller descriptions of these variables and of other embellishments of the command `solve` will be displayed in response to the command `?solve`.

### 14.4.1 Classical Turning Points

The turning points for the potential energy in Eq. (14.3) are readily found symbolically with the statements

```maple
> V := x^3/10000 + x^2/200 - x/500 - 1/2:
> soln := solve(V=0, x);
soln := mess, mess, mess
```

where the indicated messes, which—because of the presence of $i = \sqrt{-1}$ here and there—will appear superficially to be complex, are too long and too unfathomable to reproduce here. We can, however, convert this output directly to a (complex) floating point form with the statement\(^6\)

```maple
> evalf(soln);
9.34870420 + 0.1, -48.26826758 + 67949192 \times 10^{-10} I,
-11.08043664 - 3.732050808 \times 10^{-9} I
```

thereby reducing the messes to quantities that are more easily interpreted. The real parts of the roots here obtained are certainly consistent with the bounds given in Eq. (14.4). Further, the assertion that the imaginary parts of the roots are all zero is not unreasonable, given that, by default, the precision of the command `evalf` is on the order of one in the tenth digit. Indeed, we can support this understanding of the imaginary part by evaluating the roots to more decimal places with the statements

```maple
> Digits := 20:
> evalf(soln);
9.34870419760052270 - 1. \times 10^{-18} I,
-48.26826756675163413 - 7.392304845413263761 \times 10^{-19} I,
-11.08043663084841814 + 1.3392304845413263761 \times 10^{-18} I
```

```maple
> Digits := 10:
```

Even at this precision, the assertion that the imaginary parts are all zero is not unreasonable, given that the precision of the command `evalf` is on the order of one in the twentieth digit when the environment variable `Digits` has the value 20.

---

\(^6\)Remember that really small values obtained in this way are subject to significant computer round-off and may well differ significantly from version to version and platform to platform.
14.4. Equilibrium Points

Finding the equilibrium points for the potential energy of Eq. (14.3) is easier. We proceed from the above dialog with the statements

\[
\begin{align*}
> & \text{dVdx := diff( V, x ); } \\
> & \text{solve( dVdx = 0, x ); } \\
& \quad -\frac{50}{3} + \frac{16}{3}\sqrt{10}, -\frac{50}{3} - \frac{16}{3}\sqrt{10}
\end{align*}
\]

\[
> \text{soln := evalf( % ); } \\
& \quad 0.19881418, -33.53214752
\]

finding first \( dV/dx \) and then finding the values of \( x \) at which \( dV/dx = 0 \). These results are consistent with the bounds obtained in Eq. (14.7). Finally, to assess the stability of each equilibrium, we evaluate the second derivative at each equilibrium point with the statements

\[
\begin{align*}
> & \text{d2Vdx2 := diff( dVdx, x ); } \\
> & \text{eval( d2Vdx2, x=soln[1] ); } \\
& \quad 0.01011928851 \\
> & \text{eval( d2Vdx2, x=soln[2] ); } \\
& \quad -0.01011928851
\end{align*}
\]

concluding that the first root \( x = 0.19881418 \) locates a stable equilibrium (\( d^2V/dx^2 > 0 \); local minimum in the potential energy) while the second root \( x = -33.53214752 \) locates an unstable equilibrium (\( d^2V/dx^2 < 0 \); local maximum in the potential energy).

14.4.3 Natural Frequencies

To find the natural frequencies for the coupled oscillators described in Section 14.1.3, we seek roots of the function \( D(\omega) \) defined in Eq. (14.12). Since \( D(\omega) \) is, in fact, quadratic in \( \omega^2 \), we solve first for \( \omega^2 \)—though we must substitute a single variable \( \lambda \) for \( \omega^2 \) before doing so—and then we take the square root of that solution with the statements\(^7\,^8\)

\[
\begin{align*}
> & \text{d := omega^4 - 2*omega^2*(1+kappa) + 1 + 2*kappa: } \\
> & \text{subs(omega=sqrt(lambda), d ): } \\
& \quad \{\text{solve( %=0, lambda )}\} \\
& \quad [1, 1 + 2\kappa] \\
> & \text{freqs := map( sqrt, % ); } \\
& \quad \text{freqs:= [1,}\sqrt{2\kappa+1}\text{]}
\end{align*}
\]

Note—as must on physical grounds be the case—that the solutions obtained for \( \lambda = \omega^2 \) are both positive, so the square roots will be real. (We ignore the negative square roots, since they don’t have the physical significance of the positive square roots.) Note also that the first of the two frequencies is the same for all \( \kappa \) but that the second increases steadily as \( \kappa \)—the strength of the coupling—increases.

To make the point at the end of the last paragraph even more explicitly, we generate a graph of the two frequencies as functions of \( \kappa \) with the statement

\[\text{At this point, for simplicity, we drop the overbar on } \kappa.\]
\[\text{The square brackets around the command solve result in the solution being returned as a list rather than a sequence—necessary so that the function sqrt can subsequently be mapped over the elements of the list. If the result is accepted as a sequence, the map statement involving the function sqrt will misinterpret the second item in the sequence. For details, see the help message ?sqrt.}\]
406

CHAPTER 14. FINDING ROOTS

Figure 14.6: Natural frequencies for coupled oscillators. The horizontal coordinate is $\kappa = k'/k$ and the vertical coordinate is the frequency in units of $\omega_0 = \sqrt{k/m}$.

> plot( freqs, kappa=0.0..4.0, y=0.0..3.0, color=black, thickness=3, labels=["kappa", "omega"], labeldirections=[HORIZONTAL,VERTICAL], axesfont=[TIMES,ROMAN,14], labelfont=[TIMES,ROMAN,16] );

The resulting graph is shown in Fig. 14.6.

14.4.4 Range of Projectile in Viscous Medium

While the problem of finding the range of a projectile in a viscous medium is not effectively addressed by symbolic methods, certain limits of the behavior of that projectile are readily obtained by invoking MAPLE. We begin by defining the trajectory as given by Eqs. (14.15) and (14.16) with the statements\(^9\)

> x := (m*v0*cos(theta)/b) * ( 1 - exp(-b*t/m) ):
> y := -m*g*t/b+(m/b)*(v0*sin(theta)+m*g/b) * ( 1 - exp(-b*t/m) ):

In many instances, the damping constant $b$ is small. Thus, we can sometimes be satisfied with a Taylor expansion of these quantities about $b = 0$, an expansion we obtain with the MAPLE statements

\(^9\)To facilitate visualization, some of the output in the following dialog with MAPLE has been recast from the actual form presented by MAPLE.
14.4. FINDING ROOTS SYMBOLICALLY WITH MAPLE

> xb := taylor( x, b=0, 3 );

\[\text{xb} := v_0 \cos(\theta) t - \frac{1}{2} \frac{v_0 \cos(\theta) t^2}{m} b + O(b^2)\]

> yb := taylor( y, b=0, 4 );

\[-\frac{1}{2} g t^2 + v_0 \sin(\theta) t + \left( \frac{1}{6} \frac{g t^3}{m} - \frac{1}{2} \frac{v_0 \sin(\theta) t^2}{m} \right) b + O(b^2)\]

We notice first—reassuringly—that the terms that are zeroth-order in \(b\), i.e., \(x = v_0 t \cos \theta\) and \(y = -\frac{1}{2} gt^2 + v_0 t \sin \theta\), are exactly what we would expect for the projectile in the absence of air resistance. More generally, to find the range when \(b\) is small, we begin by converting the expression to a polynomial, removing the root at \(t = 0\), and solving for the roots of the Taylor expansion of \(y\) with the statements

> yb := convert( yb, polynom );
> yb := expand( yb/t );
> soln := solve( yb=0, t );

\(\text{soln} := \text{mess, mess}\)

To sort out which of these two roots is the physically significant one, we evaluate the limit of this expression as \(b \to 0\) with the statements

> interface( showassumed=0 );
> assume( b>0, m>0, g>0 );
> series( soln[1], b=0, 2 );

\[\frac{3}{b} \frac{m}{b} + \frac{v_0 \sin \theta}{g} + O(b)\]

> series( soln[2], b=0, 2 );

\[\frac{2}{b} \frac{v_0 \sin \theta}{g} + O(b)\]

Since the first root becomes arbitrarily large as \(b\) becomes small, we must reject it on physical grounds. Only the second root, which we can display more fully with the statement

> expand( soln[2] );

\[\frac{3}{2} \frac{m}{b} \frac{v_0 \sin(\theta)}{g} - \frac{1}{2} \frac{v_0^2 m^2 - 6v_0 bg \sin \theta + 9b^2 v_0^2 \sin^2 \theta}{bg}\]

is of interest. Finally, for consistency with approximations already made, we find this root to first order in \(b\) with the statement

> tim := taylor( soln[2], b=0, 3 );

\[\text{tim} := \frac{2}{3} \frac{v_0 \sin \theta}{g} - \frac{2}{3} \frac{v^2_0 \sin^2 \theta}{g^2 m} b + O(b^2)\]

and find the range by substituting this result into the first order expression for the horizontal coordinate with the statements
\[ \text{rang} := \text{eval} ( \text{convert}(xb, \text{polynom}), t=\text{convert}(\text{tim}, \text{polynom}) ) ; \]
\[ \text{rang} := \text{taylor} ( \text{rang}, b=0, 2 ) ; \]
\[ \text{rang} := \text{convert} ( \text{rang}, \text{polynom} ) ; \]
\[ \text{rang} := \frac{2v_0^2}{g} \sin \theta \cos \theta \left( 1 - \frac{4v_0b}{3mg} \sin \theta \right) \]

Note in particular that, in the absence of viscous resistance \((b = 0)\), this result reduces to the known result \((v_0^2/g) \sin 2\theta\).

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
\[ x_{\text{mid}} = \frac{1}{2} (x_{\text{min}} + x_{\text{max}}) \quad \text{and} \quad f_{\text{mid}} = f(x_{\text{mid}}) \quad (14.34) \]
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. 14.7. 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
Figure 14.7: The method of bisection. The two parts show the positioning of critical points (a) when the function increases as $x$ increases and (b) when the function decreases as $x$ increases.

Figure 14.8: Newton’s method. The intersection of a tangent line drawn to the curve at the current estimate of the root with the horizontal axis provides the next estimate of the root.

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

$$\frac{\Delta f}{\Delta x} = \frac{0 - f_n}{x_{n+1} - x_n} = f'_n \quad \Rightarrow \quad x_{n+1} = x_n - \frac{f_n}{f'_n}$$

(14.35)

The relevant geometry is illustrated in Fig. 14.8. 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 second in Eq. (14.29), and finally takes as \(x_4\)—which replaces \(x_1\) 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

\[
f(x) = \prod_{i=1}^{n}(x - x_i)
\]

(14.36)

(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

\[
x_{n+1} = x_n - \frac{N}{G \pm \sqrt{(N-1)(NH-G^2)}}
\]

(14.37)

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 \(G = f'/f\) 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 \(f(x_{\text{root}}) = 0.0001\), 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.\(^{10}\) 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 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

\(^{10}\)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.
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.

### 14.7 Finding Roots Numerically with IDL

**Note:** All IDL program (.pro) files referred to in this chapter are available in the directory `$HEAD/idl`, where (as defined in the Local Guide) `$HEAD` must be replaced by the appropriate path for your site. At some sites, this directory or some other directory containing these files may also have been placed in IDL’s default search path. If so, the files can be found by IDL without explicit specification of a path. Otherwise, you will have to use the full path to copy them into your default directory to access them.

Root-finding functions in IDL include

- **fz_roots**, which uses Laguerre’s method to find the roots of a polynomial, whose coefficients may be complex. It is invoked with a statement like

  \[
  q = \text{fz_roots}( c, \text{double}, \text{eps}=\text{value})
  \]

  where \(c\), which is mandatory, is a (possibly complex) vector containing the coefficients of the terms in the polynomial (constant first), and \(q\) is a complex vector storing the returned roots. If included, the optional keyword \text{double} will select a double-precision calculation; the optional keyword \text{eps} (default \(2 \times 10^{-6}\)) specifies the desired fractional accuracy.

- **fx_root**, which uses Müller’s method to find real and complex roots of nonlinear functions of a single variable. It is invoked with a statement like

  \[
  q = \text{fx_root}( x, 'func', \text{double}, \text{itmax}=\text{value}, \text{tol}=\text{value})
  \]

  where \(x\), which is mandatory, is a three-element vector containing initial guesses, \text{func}—also mandatory—is a string giving the name of a function pro-file that returns the value of the function whose roots are sought, and \(q\) stores the returned root. If included, the optional
keyword \texttt{double} will select a double-precision calculation; the optional keyword \texttt{itmax} (default 100) specifies the maximum number of iterations and the optional keyword \texttt{tol} (default $10^{-4}$) specifies the convergence criterion by stipulating the largest acceptable difference between consecutive iterates.

- \texttt{newton}, which uses Newton’s method to determine a root of a nonlinear function of a single variable. It is invoked with a statement like

\begin{verbatim}
q = newton( x, 'func', /double, itmax=value, tolf=value, tolx=value)
\end{verbatim}

where \texttt{x}, which is mandatory, provides the initial guess for the desired root, \texttt{func}—also mandatory—is a string giving the name of a function pro-file that returns the value of the function whose roots are sought, and \texttt{q} stores the returned root. If included, the optional keyword \texttt{double} will select a double-precision calculation; the optional keyword \texttt{itmax} (default 200) specifies the maximum number of iterations; the optional keyword \texttt{tolf} (default $10^{-4}$) specifies how close to zero the value of the function at the root is to be before stopping; and \texttt{tolx} (default $10^{-7}$) stipulates the largest acceptable difference between consecutive iterates. Other optional keywords are described in the IDL manuals.

In addition, Section 14.A contains a listing of the Lawrence-written program \texttt{lubisect.pro}, which defines an IDL function that uses the method of bisection. It is called with a statement of the form

\begin{verbatim}
q = lubisect( 'func', xlb, xub, tolf=value, itmax=value)
\end{verbatim}

where \texttt{func}, which is mandatory, is a string giving the name of a function pro-file that returns the value of the function whose roots are sought, \texttt{xlb} and \texttt{xub}—also mandatory—give the lower and upper bounds on an interval containing the desired root, and \texttt{q} stores the returned root. If included, the optional keyword \texttt{tol} (default $10^{-6}$) stipulates the largest acceptable difference between consecutive iterates; the optional keyword \texttt{itmax} (default 20) specifies the maximum number of refinements before the search will terminate in case convergence as specified by \texttt{tol} fails. As the routine is entered, it checks to make sure the function has different signs at the specified end points and, if not, returns to the calling program immediately, setting \texttt{q} equal to the entered value of \texttt{xlb}.

### 14.7.1 Classical Turning Points

We might, for example, find the turning points for the potential energy in Eq. (14.3) with the IDL statements

\begin{verbatim}
IDL> A = [ -0.5, -1.0/500.0, 1.0/200.0, 1.0/10000.0]
IDL> roots = fz_roots( A )
IDL> print, roots
(-48.2683, 0.00000)(-11.0804, 0.00000)( 9.34870, 0.00000)
\end{verbatim}

Even though they have been presented as complex numbers, these roots have turned out to be real (zero imaginary part). For later manipulations, we might actually prefer to have the roots in a three-component real vector rather than a three-component complex vector. IDL’s command \texttt{float}, which converts other data types to floating point representation, is built so that the conversion of a complex value to a floating value is equivalent to extracting the real part of the complex value. Thus, the simple statements

\begin{verbatim}
(-48.2683, 0.00000)(-11.0804, 0.00000)( 9.34870, 0.00000)
\end{verbatim}
14.7. FINDING ROOTS NUMERICALLY WITH IDL

IDL> roots = float(roots)
IDL> print, roots
   -48.2683  -11.0804  9.34870

extract the real parts alone, store them in a simple real vector, and display that vector.

Other methods for finding the required roots all require that the function be defined in a pro-file. For this example, we might construct the file

```
function turnpts, x
return, x^3/10000.0+x^2/200.0-x/500.0 - 0.5
end
```

which we store in the default directory with the name `turnpts.pro`. Then, we would find the roots of the defined polynomial with the IDL statements

```
IDL> trial = [ -60.0, -50.0, -40.0 ]
IDL> root = fx_root( trial, 'turnpts' )
IDL> print, root
   -48.2683

IDL> trial = [ -50.0, -40.0, -30.0 ]
IDL> print, fx_root( trial, 'turnpts' )
   -11.0804

IDL> trial = [ 10.0, 20.0, 30.0 ]
IDL> print, fx_root( trial, 'turnpts' )
   9.34870
```

For this method, we needed some information about the location of each root in order to start the process, and we based our input on the information extracted from Fig. 14.1.

The same function pro-file can alternatively be used in an invocation of `newton` with the statement

```
IDL> print, newton( -50.0, 'turnpts' )
   -48.2683
```

or in an invocation of `lubisect` with the statement

```
IDL> print, lubisect( 'turnpts', -50.00, -47.50 )
   -48.2683
```

With different starting values, the routines `newton` and `lubisect` would, of course, converge on the remaining roots.

14.7.2 Natural Frequencies

Because the natural frequencies of the coupled oscillator described in Section 14.1.3 are roots of a polynomial [Eq. (14.12)], we invoke `fz_roots`. We seek the roots as functions of \( \kappa \), which we shall represent by the variable `kvals` in our statements to IDL. To set a definite objective, we suppose that, ultimately, we would like to have a graph of each of the two roots as a function of \( \kappa \) over the interval \( 0.0 \leq \kappa \leq 4.0 \), and we elect to divide that interval into 40 steps of length 0.1 each. The coefficients of the polynomial are, of course, different for each value of \( \kappa \). Let us therefore begin by creating two arrays, the first of which—a column vector with 41 elements—contains the values of \( \kappa \) and the second of which—an array with 3 columns and 41 rows—contains in each row the coefficients of the polynomial (seen as a quadratic polynomial in \( \omega^2 \)) for the corresponding value of \( \kappa \). These ends are achieved with the statements
Table 14.1: The IDL function projectile.pro.

function projectile, tau
common params, alpha, theta ; Provide for passing parameters
tmp = 1.0+alpha*sin(theta) ; Calculate coefficient
return, tau - tmp*(1-exp(-tau)) ; Return value of f(tau)
end

IDL> kvals = transpose( findgen(41)/10.0 )
IDL> ones = transpose( 1.0 + fltarr(41) )
IDL> coeffs = [ 1.0+2.0*kvals, -2.0*(1.0+kvals), ones ]

which set the values of \( \pi \), create a 41-element column vector of ones, and then create the \( 3 \times 41 \) array of coefficients. Then, we find the roots and concatenate them into a single array with the statements

IDL> roots = fz_roots( coeffs[*,0] )
IDL> for i = 1, 40 do roots = [ [roots], [fz_roots( coeffs[*,i] )] ]

The result is a 2 column \( \times 41 \) row array of complex values, the first column containing the first root for each value of \( \pi \) and the second column containing the second root. As we suspect (and as the statement print, roots will confirm), these roots are all real; their imaginary parts are all zero. Thus, we extract the only physically significant part—the real part—with the statement

IDL> roots = float( roots )

Finally, since the roots we have at the moment are values of \( \omega^2 \), we take the square root to find the values of \( \omega \) and then plot a graph of each of the roots—one contained in the first column of the array roots and the other contained in the second column—with the statements

IDL> roots = sqrt( roots )
IDL> plot, kvals, roots[0,*], ticklen=1.0, thick=3.0, yrange=[0.0,3.0]
IDL> oplot, kvals, roots[1,*], thick=3.0

The resulting graph is shown in Fig. 14.9.

14.7.3 Range of Projectile

As laid out in Section 14.1.4, finding the range of a projectile moving in a viscous medium begins with finding the time at which a projectile launched at some angle \( \theta \) in a medium characterized by a (dimensionless) viscous damping coefficient \( \alpha = b v_0 / (m g) \) returns to its initial altitude. In other words, we must find the one non-zero root of the function \( f(\pi) \) defined in Eq. (14.18). Central to the entire calculation is the function pro-file listed in Table 14.1, which we store in the default directory in a file named projectile.pro. Indirectly, we have already used this function, since, for example, in creating Fig. 14.4, we set various preliminary variables with the statements

function projectile, tau
common params, alpha, theta ; Provide for passing parameters
tmp = 1.0+alpha*sin(theta) ; Calculate coefficient
return, tau - tmp*(1-exp(-tau)) ; Return value of f(tau)
end
Figure 14.9: Natural frequencies as a function of coupling strength. Two normal modes exist. For one, the natural frequency is constant; for the other, the natural frequency increases as the coupling strength increases.

IDL> common params, alpha, theta
IDL> thetavals = !pi * findgen(7)/12.0
IDL> !p.ticklen=1.0 & !p.thick = 3.0
IDL> !p.multi=[0,2,2] & !p.charsize=0.8
IDL> tau=findgen(201)/100.0

Provide for passing parameters.
Set \( \theta = 0^\circ, 15^\circ, \ldots, 90^\circ \) in radians.
Set ticklength; set line thickness.
Set 2 \times 2 \) display; set character size.
Set values of \( \tau \).

Then, we produced the upper left frame with the statements

IDL> alpha = 0.1
IDL> theta = thetavals[0]
IDL> plot, tau, projectile(tau), $  
IDL> xrange=[0.0,0.25], $  
IDL> yrange=[-0.005,0.005], $  
IDL> title='!4a!3=0.1', xtitle='!4s!3', $  
IDL> ytitle='f(!4s!3)'  
IDL> for i=1,6 do begin $  
IDL> theta = thetavals[i] & $  
IDL> oplot, tau, projectile(tau) & $  
IDL> endfor

Choose \( \alpha \).
Plot graph for \( \theta = 0^\circ \).
Overplot graphs for \( \theta = 15^\circ, \ldots, 90^\circ \).

For this example, we choose IDL's built-in routine newton.\textsuperscript{11} Starting with a fresh IDL session, choosing \( \alpha = 0.4 \), setting \( \theta = 30^\circ \) (in radians), referring to Fig. 14.4 to support the initial guess \( \tau = 0.5 \), and accepting (for the moment) all defaults, we find the corresponding time of return to the initial altitude by invoking the statements

\textsuperscript{11}The function of interest is not a polynomial, so fz_roots is not an option. The routines fx_root and lubisect would work, but they require three and two initial guesses, respectively. Since we want to find many roots, our task will be simplest if we are obliged to provide only one initial guess.
IDL> common params, alpha, theta

IDL> alpha = 0.4 & theta = !pi*30.0/180.0

IDL> tau0 = 0.5

IDL> root = newton( tau0, 'projectile' )

IDL> print, root

0.376439

The corresponding range is then given by Eq. (14.19), which we evaluate and display for the specific case of this paragraph with the statements

IDL> range = cos(theta) * ( 1-exp(-root) ) / alpha

IDL> print, range

0.679178

This result is expressed in units of \( v_0^2/g \).

To find the angle of fire to attain maximum range, we need to repeat the sample calculation of the previous paragraph for values of \( \theta \) ranging from 0\(^\circ\) to 90\(^\circ\), choosing, say increments of 1\(^\circ\) between consecutive values of \( \theta \). Continuing in the same session with IDL, we prepare for a calculational loop with the statements

IDL> thetadeg = findgen(91)

IDL> thetarad = !pi*thetadeg/180.0

IDL> roots = fltarr(91)

Then, choosing \( \alpha = 0.1 \) and noting from the first frame in Fig. 14.4 that all roots lie below \( \tau = 0.2 \), we continue with the statements\(^{12}\)

IDL> alpha = 0.1 & tau0 = 0.2

IDL> for i = 0, 90 do begin

IDL> theta = thetarad[i] & roots[i] = newton( tau0, 'projectile', $

IDL> /double, tolf=1e-8, tolx=1e-10 ) & $

IDL> endfor

IDL> range1 = cos(thetarad)*(1-exp(-roots))/alpha

IDL> plot, thetadeg, range1, thick=3.0, ticklen=1.0, $

IDL> xtitle='\pi h\textnormal{ (deg)}', ytitle='R(\pi h\textnormal{)}'

Repeating this process with (\( \alpha = 0.2, \tau_0 = 0.4 \)), with (\( \alpha = 0.4, \tau_0 = 0.8 \)), and with (\( \alpha = 0.8, \tau_0 = 1.5 \)), storing the results in \texttt{range2}, \texttt{range4}, and \texttt{range8}, respectively, and overplotting each on the axes created above, we finally produce Fig. 14.10. As \( \alpha \) increases (\( b \) increases, \( v_0 \) increases, \( m \) decreases, or some combination), the maximum range (measured in units of \( v_0^2/g \)) decreases and the angle of fire to achieve that range becomes shallower and shallower.

Even more specifically, we can examine the values in the four variables \texttt{range} to find the maximum range and the approximate firing angle to achieve it in each case. For example, the statements

IDL> print, max(range1)

0.913793

IDL> print, range1[43], range1[44], range1[45]

0.913564 0.913793 0.912932

\(^{12}\)A bit of exploration was needed before choosing the tolerances specified in the call to \texttt{newton}. As might be expected from the appearance Fig. 14.4, particular difficulties with accuracy appeared for smaller values of \( \alpha \) and especially for small values of \( \theta \), where the zero and non-zero roots are quite close together.
14.7. FINDING ROOTS NUMERICALLY WITH IDL

Figure 14.10: Range versus $\theta$ for $\alpha = 0.1$ (highest graph), $\alpha = 0.2$, $\alpha = 0.4$, and $\alpha = 0.8$ (lowest graph).

![Graph of range versus theta for different values of alpha](image)

Table 14.2: The IDL functions qmupper.pro and qmlower.pro.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>function qmupper, s</td>
<td>Name function and variable</td>
</tr>
<tr>
<td>return, sin(s) - 0.04*s</td>
<td>return value with upper sign</td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
<tr>
<td>function qmlower, s</td>
<td>Name function and variable</td>
</tr>
<tr>
<td>return, sin(s) + 0.04*s</td>
<td>Return value with lower sign</td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
</tbody>
</table>

reveal the maximum value of the range for $\alpha = 0.1$ and, further, reveal that that maximum range occurs when the firing angle is near $44^\circ$. Fitting a parabola to the three points and finding its maximum yields the angle $43.77^\circ$. Similar statements applied to the other three values of $\alpha$ yield that the maximum range and firing angle are $0.841218$ and about $43^\circ$ ($42.50^\circ$) for $\alpha = 0.2$, $0.725949$ and about $40^\circ$ ($40.35^\circ$) for $\alpha = 0.4$, and $0.569568$ and about $37^\circ$ ($37.00^\circ$) for $\alpha = 0.8$, respectively.

14.7.4 Energy Levels in Quantum Well

Finally, we turn to addressing the energy levels for the finite-depth quantum well discussed in Section 14.1.5. We seek solutions to the equations in Eq. (14.25), so we begin by creating the function pro-file qmlevels.pro containing the lines listed in Table 14.2. Since we have put two routines in a single file, we must begin our search for the actual roots by explicitly compiling qmlevels. Thus, we prepare for the task at hand with the statements

---

13See an exercise towards the end of the chapter(s) on specific symbol manipulating programs.
IDL> .compile qmlevels
Compile function pro-files.
IDL> upper = fltarr(8)
Prepare vector for roots with upper sign.
IDL> lower = fltarr(9)
Prepare vector for roots with lower sign.

Then, taking the limits as given in Eqs. (14.26) and (14.27), we find the roots in each category with the IDL statements

IDL> upper[0] = lubisect( 'qmupper', -0.5, 0.5 )
IDL> upper[1] = lubisect( 'qmupper', 2.5, 3.5 )
IDL>
IDL> upper[7] = lubisect( 'qmupper', 20.5, 21.5 )
IDL> lower[0] = lubisect( 'qmlower', -0.5, 0.5 )
IDL> lower[1] = lubisect( 'qmlower', 3.0, 4.0 )
IDL>
IDL> lower[8] = lubisect( 'qmlower', 23.5, 24.5 )

and display them with the statements

IDL> print, upper
0.00000 3.02048 6.54821 9.05418 13.1188 15.0614
19.7611 20.9943

IDL> print, lower
0.00000 3.27289 6.03920 9.82884 12.0628 16.4248
18.0433 23.1778 23.8645

Remember, however, that physically acceptable solutions must satisfy not Eq. (14.25), which we have solved, but Eq. (14.23). To determine which of these roots are physically meaningful, we substitute each into Eq. (14.23), remembering that \( c = 25 \) for the case we have treated. The appropriate statements to IDL are\textsuperscript{14}

IDL> print, upper*cos(upper)/sin(upper) + sqrt(25.0^2-upper^2)
NaN -0.000127792 48.2543 6.48499e-05 42.5627 7.62939e-06
30.6267 2.47955e-05

IDL> print, lower*cos(lower)/sin(lower) + sqrt(25.0^2-lower^2)
NaN 49.5696 -9.15527e-05 45.9736 -1.90735e-06 37.6949
-5.72205e-05 18.7393 1.81198e-05

Only those roots of Eq. (14.25) that give zero (within the precision of our determination of the root) on substitution into Eq. (14.23) can be accepted. We therefore must reject as spurious the values upper[2], upper[4], upper[6], lower[1], lower[3], lower[5], and lower[7]. We also reject upper[0] and lower[0]; they correspond to evaluation of Eq. (14.23) at \( s = 0 \) and, because \( \lim_{s \to 0} s \cot s = 1 \) while \( -\sqrt{25^2-s^2} = -25 \) when \( s = 0 \), these roots don’t satisfy the original equation. Thus, we can assemble a single vector containing the physically meaningful roots with the IDL statements

IDL> s = lower[1:8]
Discard first root in lower.
IDL> for i = 1, 7, 2 do s[i-1]=upper[i]
Replace “bad” roots in \( s \) with “good” ones in upper.

IDL> print,s
20.9943 23.8645

Finally, remembering that we are interested not so much in the values of \( s \) but in the associated energies, we exploit the equation \( E/V_0 = -(1-s^2/c^2) \) to find the allowed energies with the statements

\textsuperscript{14}The value ‘NaN’ for the first root in each category arises because of a division by zero in evaluating the function.
14.12 Finding Roots Numerically with MAPLE

When MAPLE is unable to find an analytic solution for the roots of an equation of interest, we must resort to numerical methods. MAPLE’s workhorse for this purpose is the command `fsolve`, which invokes a variation of Newton’s method. Its syntax is essentially identical to that used with the command `solve`. In its simplest form, we merely invoke the function with a specification of the equation `Eqn` to be solved and the variable `Var` to be found, for example,

```
fsolve( Eqn, Var )
```
More specifically, the statement

```maple
> fsolve( cos(x)=0, x );
1.570796327
```
returns a value we recognize as \( \pi/2 \). If \( \text{Eqn} \) is a polynomial, \textit{fsolve} returns all \textit{real} roots.\(^{15} \) For non-polynomials, \textit{fsolve} searches for a single root. In the form so far illustrated, the command makes its own judgment about the interval in which to search. We can, however, stipulate the interval in which \textit{fsolve} is to search for a root by specifying an additional argument, e.g.,

```maple
> fsolve( cos(x)=0, x, Pi..2*Pi );
4.712388980
```
a value we recognize as \( 3\pi/2 \). This additional argument provides bounds for the search and permits us to select a particular root from the many that we know \textit{a priori} exist. In all cases, the accuracy of the value returned is controlled by the environment variable \textit{Digits}. Note, of course, that \textit{fsolve} cannot return roots for any equation that contains variables to which no numeric value has been assigned.

Out of curiosity, let us test \textit{fsolve} with a polynomial that has a multiple root. We execute the statements

```maple
> p1 := expand( (x-1)^2*(x+3)*(x-4) );
> fsolve( p1, x );
```
Create polynomial with multiple roots.

```maple
−3.000000000, 1.0, 1.0, 4.000000000
```
Find roots with \textit{fsolve}.

Clearly, \textit{fsolve} has returned all four roots of the given polynomial even though one of the three distinct roots is a double root.

### 14.12.1 Classical Turning Points

To find the turning points for the potential energy in Eq. (14.3), we simply define the polynomial and invoke \textit{fsolve} with the statements

```maple
> V := x^3/10000+x^2/200-x/500-1/2:
> fsolve( V, x );
```
Define polynomial of Eq. (14.3).

```maple
−48.26826757, −11.08043663, 9.348704198
```
Find roots.

These results are certainly consistent with the bounds identified in Section 14.1.1.

### 14.12.2 Range of Projectile

As laid out in Section 14.1.4, finding the range of a projectile moving in a viscous medium begins with finding the time at which a projectile launched at some angle \( \theta \) in a medium characterized by a (dimensionless) viscous damping coefficient \( \alpha = bv_0/(mg) \) returns to its initial altitude. In other words, we must find the one non-zero root of the function \( f(\tau) \) defined in Eq. (14.18). Finding that root and then the range for a particular choice of \( \alpha \) and \( \theta \) is quickly accomplished in MAPLE with the statements

```maple
> rang := cos(theta)*(1-exp(-tau))/alpha;
> fct := tau - (1+alpha*sin(theta))*(1-exp(-tau));
```
Define range [Eq. (14.19)].

```maple
rang := \frac{\cos(\theta) (1 - e^{-\tau})}{\alpha}
```
Define function [Eq. (14.18)].

```maple
fct := \tau - (1 + \alpha \sin(\theta))(1 - e^{-\tau})
```

\(^{15}\)If the option \textit{complex} is added to the argument of \textit{fsolve}, the command will return complex roots of the polynomial as well.
14.12. FINDING ROOTS NUMERICALLY WITH MAPLE

> alpha := 0.4: theta := Pi*30.0/180.0:
> fsolve( fct, tau, 0.2..0.5 );
\[0.3764379973\]

Choose \(\alpha\), \(\theta\).

Find root.

> evalf( eval( rang, tau=%) );
\[0.6791768096\]

Evaluate range.

This result is expressed in units of \(v_0^2/g\).

To find the angle of fire to attain maximum range, we need to repeat the sample calculation of the previous paragraph for values of \(\theta\) ranging from 0\(^\circ\) to 90\(^\circ\), choosing, say, increments of 1\(^\circ\) between consecutive values of \(\theta\). To this end, we define an appropriately sized array and then accomplish all of these operations in a single loop with the statements

\[\begin{align*}
> &\text{rng := Array( 1..91 ):} \\
> &\text{for i from 0 to 90 do} \\
> &\quad \theta := \text{Pi} \times \text{i}/180.0: \\
> &\quad \text{tmp := fsolve( fct, tau, 0.005..0.8 ):} \\
> &\quad \text{rng}[\text{i}+1] := [ \text{i}, \text{evalf( eval( rang, tau=tmp ) ) } ]: \\
&\text{end do:}
\end{align*}\]

(Remember that \(\alpha\) was set to 0.4 in the previous paragraph.) The range within which roots are sought has been set with reference to Fig. 14.4, with the lower limit set so that the root \(\tau = 0\) will not be found for any of the angles. Unfortunately, the only root for \(\theta = 0\) is \(\tau = 0\). Thus, to have the correct values in all elements of \(\text{rng}\), we must execute the statement

\[\text{> rng[1]} := [0,0.0]:\]

after the above loop has terminated. Finally, we produce the graph in Fig. 14.12 with the statements

\[\begin{align*}
> &\text{rng1 := convert( rng, list ):} \\
> &\text{plots[listplot( rng1, color='black', labels=['theta','R(theta)'],} \\
> &\quad \text{thickness=3, labelfont=[TIMES,ROMAN,16], axesfont=[TIMES,ROMAN,14],} \\
> &\quad \text{labeldirections=[HORIZONTAL,VERTICAL] );}
\end{align*}\]

Further, by noting the three values,

\[\text{> [ rng1[40][2], rng1[41][2], rng1[42][2] ] ;}\]
\[\{0.7251742824, 0.7258889382, 0.7259485662\}\]

we conclude that the maximum range for \(\alpha = 0.4\) occurs near\(^{16}\) \(\theta = 40^\circ\) (or, with parabolic interpolation to find the maximum, \(\theta = 40.43^\circ\)), rather lower than the 45\(^\circ\) angle at which that maximum range occurs when \(\alpha = 0\) (no viscous damping).

Had we started with some other value of \(\alpha\), of course, statements identical to those in the previous paragraph would produce a graph corresponding to that other value of the viscous damping.

\(^{16}\)Remember that the index of the two-component entries in \(\text{rng1}\) is one larger than the angle in degrees to which that entry corresponds.
14.12.3 Energy Levels in Quantum Well

Finally, we turn to addressing the energy levels for the finite-depth quantum well discussed in Section 14.1.5, seeking solutions to the equations in Eq. (14.25). In preparation, we first creating lists of the lower and upper bounds given in Eqs. (14.26) and (14.27). To that end we execute the statements\textsuperscript{17}

\begin{verbatim}
> ulb := [-0.5, 2.5, 6.0, 8.5, 12.5, 14.5, 19.0, 20.5]:
> uub := [ 0.5, 3.5, 7.0, 9.5, 13.5, 15.5, 20.0, 21.5]:
> llb := [-0.5, 3.0, 5.5, 9.5, 11.5, 16.0, 17.5, 22.5, 23.5]:
> lub := [ 0.5, 4.0, 6.5, 10.5, 12.5, 17.0, 18.5, 23.5, 24.5]:
\end{verbatim}

Then, we find the roots in each category with the simple loops

\begin{verbatim}
> rootu := Array( 1..8 ): rootl := array( 1..9):
> for i from 1 to 8 do
>     rootu[i] := fsolve( sin(s)-0.04*s, s, ulb[i]..uub[i] ):
> end do:
> for i from 1 to 9 do
>     rootl[i] := fsolve( sin(s)+0.04*s, s, llb[i]..lub[i] ):
> end do:
\end{verbatim}

Next, we concatenate the elements into lists and display them with the statements

\begin{verbatim}
> rootu := Array( 1..8 ): rootl := array( 1..9 ):
> for i from 1 to 8 do
>     rootu[i] := fsolve( sin(s)-0.04*s, s, ulb[i]..uub[i] ):
> end do:
> for i from 1 to 9 do
>     rootl[i] := fsolve( sin(s)+0.04*s, s, llb[i]..lub[i] ):
> end do:
\end{verbatim}

\textsuperscript{17}We use a three-character variable name, where the first character will be \textit{u} or \textit{l} for the upper or lower sign in Eq. (14.25) and the remaining characters will be \textit{lb} or \textit{ub} for the lower or upper bounds on the root. Thus, for example, \textit{ulb} will contain the lower bounds for the roots associated with the upper sign.
> rtu := convert( rootu, list );

> rtl := convert( rootl, list );
rtl := [0., 3.272884921, 6.039203770, 9.828836697, 12.06284802, 16.42478429, 18.04325707, 23.17777839, 23.86449494]

Remember, however, that physically acceptable solutions must satisfy not Eq. (14.25), which we have solved, but Eq. (14.23). To determine which of these roots are physically meaningful, we substitute each into Eq. (14.23), remembering that \( c = 25 \) for the case we have treated. The appropriate statements to MAPLE are:

> f := x -> x*cot(x)+sqrt(25^2-x^2):
> map( f, rtu );
[Float(undefined), 9. \times 10^{-8}, 48.25436814, -1. \times 10^{-8}, 42.56276498, -2.0 \times 10^{-7}, 30.62674473, -1.4 \times 10^{-7}]

> map( f, rtl );
[Float(undefined), 49.56967714, -4. \times 10^{-8}, 45.97364326, 2.5 \times 10^{-7}, 37.69490481, 3. \times 10^{-8}, 18.73932650, -1.35 \times 10^{-7}]

Only those roots of Eq. (14.25) that give zero (within the precision of our determination of the root) on substitution into Eq. (14.23) can be accepted. We therefore must reject as spurious the values rtu[1], rtu[3], rtu[5], rtu[7], rtl[1], rtl[2], rtl[4], rtl[6], and rtl[8]. Thus, we assemble a single list containing the eight physically meaningful roots with the MAPLE statements:

> s := Array( 1..8 );
> for i from 1 to 8 do
    s[i] := rtl[i+1]: end do;
> for i from 1 by 2 to 7 do
    s[i] := rtu[i+1]: end do;
> print(s);

Create array for “good” roots.
Discard first root in rtl.
Replace “bad” roots in s with “good” ones in rtu.
Display “good” roots.


Finally, remembering that we are interested not so much in the values of \( s \) as in the associated energies, we exploit the equation \( E/V_0 = -(1-s^2/c^2) \) to find the allowed energies with the statement:

> energy := map( x -> -(1 - x^2/25.0^2), s );
energy := [-.9854027435, -.9416448285, -.8688347505, -.7671803162, -.6370472906, -.4791053989, -.2947838997, -.0887774101]

Then, the statement
CHAPTER 14. FINDING ROOTS

Figure 14.13: Energy level diagram for the one-dimensional quantum well when $c = 25.0$. The eight short horizontal lines in the vertical column show the energies, measured in units of $V_0$. Among other things, the grid lines show the energies of the bottom $(-1)$ and the top (0) of the well.

$$\text{plot}(\text{energy}, x=0.2..0.8, \text{color}='\text{black}', \text{view}=[0.0..1.0,-1.5..0.5], \text{thickness}=3, \text{labels}=['','E/V0'], \text{tickmarks}=[0,5], \text{gridlines}=\text{true}, \text{axesfont}=[\text{TIMES}, \text{ROMAN}, 14], \text{labelfont}=[\text{TIMES}, \text{ROMAN}, 16], \text{labeldirections}=[\text{HORIZONTAL}, \text{VERTICAL}]);$$

will produce the energy level diagram of Fig. 14.13.

14.14 Finding Roots Numerically with FORTRAN

*Note:* Except for files explicitly flagged as from the Numerical Recipes library, all FORTRAN programs (*.f) in this chapter can be copied from the directory $\text{$\text{HEAD}/fortran}$, where (as defined in the Local Guide) $\text{$\text{HEAD}$} \text{must be replaced by}$ the appropriate path for your site.

FORTRAN 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.

14.14.1 FORTRAN Programs from Scratch

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
Table 14.3: The FORTRAN program GUESSROOT.F.

**PROGRAM GUESSROOT**

WRITE(*, '(1X, A)') 'First guess ='
READ(*,*) X

DO WHILE (X .NE. 9999)
  F = x**3/10000.0 + x**2/200.0 - x/500.0 - 0.5
  WRITE(*, '(1X, A, F12.6)') 'F(X) = ', F
  WRITE(*, '(1X, A)') 'Next guess ='
  READ(*,*) X
ENDDO
END

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 14.3. After storing this program in the file guessroot.f, we compile, link, and run it with the two statements\(^{18}\)

```
f77 -o guessroot.xf guessroot.f
./guessroot.xf
```

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”\(^{19}\)

```
./guessroot.xf
  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 = -48.375 F(X) = -0.022978
  Next guess = -48.3125 F(X) = -0.009496
  Next guess = -48.2813 F(X) = -0.002794
  Next guess = -48.2656 F(X) =  0.000573
  Next guess = -48.2656 F(X) = -0.001121
  Next guess = -48.2695 F(X) = -0.000264
  Next guess = -48.2676 F(X) =  0.000144
  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,

\(^{18}\)To 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.

\(^{19}\)The prompts, entered values, and returned values have been displayed on the same line to save space.
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 FORTRAN 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 `bisect.f`, is presented in Section 14.B. This program (1) defines the function whose roots will be sought with the lines

```fortran
FUNCTION FUNC(X)
    FUNC = X**3/10000.0 + X**2/200.0 - X/500.0 - 0.5
END
```

(2) sets the tolerance and reads the initial bounds for the root with the lines

```fortran
TOL = 0.0001
WRITE(*, '(1X, A)' ) 'Lower bound = '
READ(*,*) XA
WRITE(*, '(1X, A)' ) 'Upper bound = '
READ(*,*) XB
```

and finally (3) 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

```fortran
FA = FUNC(XA)
FB = FUNC(XB)
IF ( FA*FB .GE. 0.0 ) THEN
    WRITE(*,'(1X,A)' ) 'No root in interval'
    GOTO 99
ENDIF
FM = FA
DO WHILE ( (XB-XA) .GE. TOL .AND. (FM .NE. 0.0) )
    XM = 0.5*(XB+XA)
    FM = FUNC(XM)
    IF (FA*FM .GE. 0.0) THEN
        XA = XM
    ELSE
        XB = XM
    ENDIF
END DO
WRITE(*, '(1X, A, E15.6)') 'ROOT = ', XM
WRITE(*, '(1X, A, E15.4)') 'XB-XA = ', XB-XA
WRITE(*, '(1X, A, E15.6)') 'FUNCTION = ', FM
99 END
```

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 exactly zero. Using this program, which we compile, link, and run with the statements
we have the conversation

```
./bisect.xf
Lower bound = -50.0
Upper bound = -40.0
ROOT = -0.482682E+02
XB-XA = 0.7629E-04
FUNCTION = 0.131130E-04
```

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 \( x = 9.3487 \pm 0.0001 \).

### 14.14.2 Using Numerical Recipes

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.f`, which is invoked with a statement of the form

```
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 FORTRAN driving program using `RTBIS` must be written by the user and will have the general form shown in Table 14.4. This program 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.f` to the default directory from the directory `${NRHEAD}/recipes/demo/src`, giving it the name `sinrtbis.f`, and editing it appropriately or (2) by creating the file `sinrtbis.f` with an available text editor.

2. Copy the recipe `rtbis.f` from the directory `${NRHEAD}/recipes/recipes` to the default directory.

3. Compile and link the executable module with the statement

   ```
f77 -o sinrtbis.xf sinrtbis.f rtbis.f
   ```

4. Run the program with the statement `./sinrtbis.xf`.

The lowest two and the highest non-negative roots of the sample function would then be found with the statements\(^{20}\)

\(^{20}\)To save space, prompting messages and values input are shown on the same line.
CHAPTER 14. FINDING ROOTS

Table 14.4: The FORTRAN program sinrtbis.f. A more elaborate demonstration program named xrtbis.f can be found in the directory $NRHEAD/recipes_f/demo/src.

PROGRAM SINRTBIS
EXTERNAL FUNC ! Alert compiler that FUNC is a user-defined procedure that will appear as an argument to a subroutine
WRITE(*, '(1X,A)') 'Lower bound on root = ' ! Get lower bound
READ(*,*) SLB
WRITE(*, '(1X,A)') 'Upper bound on root = ' ! Get upper bound
READ(*,*) SUB
WRITE(*, '(1X,A)') 'Error limit = ' ! Get error limit
READ(*,*) EPS
ROOT = RTBIS( FUNC, SLB, SUB, EPS ) ! Find root
WRITE(*,'(1X, A, F15.6)') 'Root is ', ROOT ! Display root
END

FUNCTION FUNC(S) ! Define function
  FUNC= SIN(S) - 0.04*S
END

An alternative algorithm uses *Newton's method* and is implemented in rtnewt.f. 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

```
./sinrtbis.xf
Lower bound on root = -0.5
Upper bound on root = 0.5
Error limit = 0.0001
Root is 0.000000

./sinrtbis.xf
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
```


\[ \text{ROOT} = \text{RTNEWT}(\text{FUNC, SLB, SUB, EPS}) \]

where \text{FUNC} defines a subroutine that returns both the function whose roots are to be found and its first derivative, \text{SLB} and \text{SUB} specify the bounds within which the desired root is known to lie, and \text{EPS} specifies the desired absolute accuracy. A suitable driving program to use \text{RTNEWT} to find the roots of our test function differs from \text{sinrthbis} in only a few respects:

- The first line would name the program \text{sinrnewt}.
- The line invoking \text{RTBIS} should invoke \text{RTNEWT} instead.
- The user-defined \textit{function} must be replaced with a \textit{subroutine} that returns both the value of the function and the value of its derivative. Thus, the function \text{FUNC} in \text{sinrthbis} must be replaced with the subroutine

```fortran
SUBROUTINE FUNC( S, F, DFDX )
    F = SIN(S) - 0.04*S
    DFDX = COS(S) - 0.04
RETURN
END
```

The steps in finding this root then are

1. Create the driving program, either (1) by copying the demonstration program \text{xrtnewt.f} from the directory \$NRHEAD/recipes_f/demo/src, giving it the name \text{sinrnewt.f} and editing it appropriately or (2) by creating the file \text{sinrnewt.f} with an available text editor.

2. Copy the recipe \text{rtnewt.f} from the directory \$NRHEAD/fortran/recipes_f/recipes to the default directory.

3. Compile and link the executable module with the statement

   \[ \text{f77 -o sinrnewt.xf sinrnewt.f rtnewt.f} \]

4. Run the program with the statement \text{./sinrnewt.xf}.

The lowest two and the highest non-negative roots of the sample function would then be found with the statements\footnote{To save space, the prompting messages and the responses are shown on the same line.}

\[ ./\text{sinrnewt.xf} \]
\[ \text{Lower bound on root} = -0.5 \]
\[ \text{Upper bound on root} = 0.5 \]
\[ \text{Error limit} = 0.0001 \]
\[ \text{Root is} \quad 0.000000 \]
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 \texttt{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 \texttt{rtbis} and \texttt{rtnewt} is left to the exercises at the end of the chapter.

Finally, we mention an auxiliary subroutine \texttt{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 \texttt{rtbis}. It is called with the statement

\begin{verbatim}
CALL ZBRAK( FUNC, A, B, N, SLB, SUB, NROOTS )
\end{verbatim}

When called, \texttt{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 \texttt{zbrak}, and then invoke \texttt{rtbis} in a loop whose index steps from one bracket to the next. The demonstration programs $\$NRHEAD$/$recipes$/$demo/src/$xrtbis$.f, $\ldots$/$xrtnewt$.f, and $\ldots$/$xrtsafe$.f all illustrate this strategy. Note that \texttt{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 \texttt{zbrak} on functions that have singularities in the interval in which roots are sought.

\section*{14.15 Finding Roots Numerically with C}

Note: Except for files explicitly flagged as from the Numerical Recipes library, all C programs (*.c) in this chapter can be copied from the directory $\$HEAD/cc$, where (as defined in the Local Guide) $\$HEAD$ must be replaced by the appropriate path for your site.

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. FINDING ROOTS NUMERICALLY WITH C

Table 14.5: 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( "%.1f", &x );
    while( x != 9999 ){
        f = pow(x,3)/10000.0 + pow(x,2)/200.0 - x/500.0 - 0.5;
        printf( "f(x) = %.15f\n", f );
        printf( "Next guess : "); scanf( "%.1f", &x ) ;
    }
}
```

14.15.1 C Programs from Scratch

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 14.5. After storing this 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
```

22To 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.

23The 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.

24To 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.

25The prompt, the entered values, and the returned values have been displayed on the same line to save space.
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 `bisect.c`, is presented in Section 14.C. 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 `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 `xa`, `xm`, `xb`; (3) sets the tolerance and reads the initial bounds for the root with the lines

```c
tol = 0.0001
printf( "Lower bound = " ); scanf( "%f", &xa );
printf( "Upper 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 `void` in order to get away with the simple statement `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 exactly zero. Using this program, which we compile, link, and run with the statements
we have the conversation

```
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

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

```
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 14.6. 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\(^{26}\) `$NRHEAD/recipes_c-ansi/demo/src`, giving it the name `sintbis.c`, and editing it appropriately or (2) by creating the file `sintbis.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`.

\(^{26}\)Alternatively, at some sites, the language directory may be `recipes_c-kr` rather than `recipes_c-ansi`. 
Table 14.6: The C program `sinrtbis.c`. A more elaborate demonstration program named `xrtbis.c` can be found in the directory `$NRHEAD recipies.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 );
}
```

5. Compile and link the executable module with the statement

```bash
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

```bash
./sinrtbis.xc
Lower bound on root = -0.5
Upper bound on root = 0.5
Error limit           = 0.0001
Root is              0.000000
```

---

27 To save space, prompting messages and values input are shown on the same line.
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

\[ \text{root}=\text{rtnewt}( \text{func}, \text{slb}, \text{sub}, \text{eps} ); \]

where \text{func} defines a procedure that returns both the function whose roots are to be found and its first derivative, \text{slb} and \text{sub} specify the bounds within which the desired root is known to lie, and \text{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 \text{func} in sinrtbis must be replaced with the procedure

```c
/* 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

```
cc -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\(^{28}\)

```
./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 `~/recipes_c-ansi/demo/src/rtbis.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.

\(^{28}\)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.\(^{29}\) 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.\(^{30}\)

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 illustrated in the exercises at the end of this chapter, Kirchoff’s laws in DC circuit theory, least squares fitting of polynomials to experimental data, and some boundary value problems are among the contexts in which linear systems—sometimes quite large linear systems—of equations arise.

Whatever their physical origin, systems of linear equations can most conveniently be presented in matrix form, e.g.,

\[
\begin{pmatrix}
  a_{11} & a_{12} & a_{13} \\
  a_{21} & a_{22} & a_{23} \\
  a_{31} & a_{32} & a_{33}
\end{pmatrix}
\begin{pmatrix}
  x_1 \\
  x_2 \\
  x_3
\end{pmatrix}
= \begin{pmatrix}
  b_1 \\
  b_2 \\
  b_3
\end{pmatrix}
\]

(14.38)

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

\[
A \mathbf{x} = \mathbf{b}
\]

(14.39)

and we would write their solution formally as

\[
\mathbf{x} = A^{-1} \mathbf{b}
\]

(14.40)

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

\[
x_1 = \frac{\begin{vmatrix}
  b_1 & a_{12} & a_{13} \\
  b_2 & a_{22} & a_{23} \\
  a_{11} & a_{12} & a_{13}
\end{vmatrix}}{\begin{vmatrix}
  a_{11} & a_{12} & a_{13} \\
  a_{21} & a_{22} & a_{23} \\
  a_{31} & a_{32} & a_{33}
\end{vmatrix}} ;
x_2 = \frac{\begin{vmatrix}
  a_{11} & b_1 & a_{13} \\
  a_{21} & a_{22} & a_{23} \\
  a_{31} & a_{32} & a_{33}
\end{vmatrix}}{\begin{vmatrix}
  a_{11} & a_{12} & a_{13} \\
  a_{21} & a_{22} & a_{23} \\
  a_{31} & a_{32} & a_{33}
\end{vmatrix}} ;
x_3 = \frac{\begin{vmatrix}
  a_{11} & a_{12} & b_1 \\
  a_{21} & a_{22} & a_{23} \\
  a_{31} & a_{32} & a_{33}
\end{vmatrix}}{\begin{vmatrix}
  a_{11} & a_{12} & a_{13} \\
  a_{21} & a_{22} & a_{23} \\
  a_{31} & a_{32} & a_{33}
\end{vmatrix}}
\]

(14.41)

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

---

\(^{29}\)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 \textit{singular} and will either have no solution (equations inhomogeneous; \( b \neq 0 \)) or an infinite number of solutions (equations homogeneous; \( b = 0 \)).

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 round off error.\(^{31}\) We can, however, invent alternative methods that are computationally more satisfactory. The simplest algorithm to describe involves \textit{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 \textit{backsubstitution}. We illustrate with Eq. (14.38), 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\(^ {32} \)
  \[
  \begin{pmatrix} 1 & a'_{12} & a'_{13} \\ 1 & a'_{22} & a'_{23} \\ 1 & a'_{32} & a'_{33} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b'_1 \\ b'_2 \\ b'_3 \end{pmatrix}
  \quad (14.42)
  \]

- keeping the first equation and replacing the second and third with the result of subtracting the first from each in turn, obtaining
  \[
  \begin{pmatrix} 1 & a'_{12} & a'_{13} \\ 0 & a''_{22} & a''_{23} \\ 0 & a''_{32} & a''_{33} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b'_1 \\ b''_2 \\ b''_3 \end{pmatrix}
  \quad (14.43)
  \]

- dividing the second and third equations by the coefficient of \( x_2 \), obtaining
  \[
  \begin{pmatrix} 1 & a'_{12} & a'_{13} \\ 0 & 1 & a''_{23} \\ 0 & 1 & a''_{33} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b'_1 \\ b''_2 \\ b''_3 \end{pmatrix}
  \quad (14.44)
  \]

- keeping the first and second equations but replacing the third with the result of subtracting the second from the third, obtaining
  \[
  \begin{pmatrix} 1 & a'_{12} & a'_{13} \\ 0 & 1 & a''_{23} \\ 0 & 0 & a''_{33} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b'_1 \\ b''_2 \\ b''_3 \end{pmatrix}
  \quad (14.45)
  \]

In essence, Gaussian elimination converts the original system of equations with a general coefficient matrix into an equivalent system whose coefficient matrix is \textit{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

\(^{31}\)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 round off error.

\(^{32}\)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) condi-
tions, 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.\(^{33}\) That is, we can write

$$A = LU$$

where

$$L = \begin{pmatrix}
1 & 0 & 0 \\
1 & 1 & 0 \\
1 & 1 & 1
\end{pmatrix} \quad \text{and} \quad U = \begin{pmatrix}
u_{11} & u_{12} & u_{13} \\
0 & u_{22} & u_{23} \\
0 & 0 & u_{33}
\end{pmatrix}$$ \hspace{1cm} (14.46)

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

$$Ax = b \implies LUx = b \implies Ly = b \quad \text{where} \quad y = Ux$$ \hspace{1cm} (14.47)

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 round off 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 that we have added the uncertainties associated with convergence to those potentially generated

\(^{33}\)The 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.
by computer round-off. 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.

MAPLE: Within MAPLE, solving systems of linear equations is accomplished with the same commands—\texttt{solve} and \texttt{fsolve}—that we have used for solving single equations symbolically and numerically. Essentially, for systems of equations, the first argument to \texttt{solve} or \texttt{fsolve} becomes a \texttt{set} containing the equations to be solved and the second argument becomes a \texttt{set} containing the variables for which solutions are sought.

IDL: For numerical solution of systems of linear equations, the most commonly used IDL commands are \texttt{cramer}, which uses Cramer's rule; \texttt{ludc} and \texttt{lusol}, which effect the LU decomposition of the coefficient matrix and the solution of the associated system; \texttt{trisol}, which is especially adapted to tridiagonal systems; and \texttt{svdc} and \texttt{svsol}, which effect the singular value decomposition of the coefficient matrix and the solution of the associated system. All of these routines use direct methods.

\textbf{NUMERICAL RECIPES in FORTRAN}: For numerical solution of systems of linear equations, the FORTRAN subroutines in the \textit{Numerical Recipes} library include \texttt{gaussj.f}, which uses Gauss-Jordan elimination (similar to Gaussian elimination); \texttt{ludcmp.f} and \texttt{lubksb.f}, which effect the LU decomposition of the coefficient matrix and the solution of the associated system; \texttt{tridag.f}, which is especially adapted to tridiagonal systems; and \texttt{svdcmp.f} and \texttt{svbksb.f}, which effect the singular value decomposition of the coefficient matrix and the solution of the associated system. All of these routines use direct methods.

\textbf{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.

\subsection*{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{equation}
\begin{align*}
f_1(x_1, x_2) &= 0 \quad ;
\end{align*}
\end{equation}
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. 14.14, for example, shows the zero contours for the two functions

\[
f_1(x_1, x_2) = \sin \left( \frac{x_1^2}{20} \right) - \cos \left( \frac{x_2}{5} \right) ; \quad f_2(x_1, x_2) = x_1 \tanh(x_2) - 5.0 \tag{14.49}
\]

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 . . . . 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, 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

\[ f_1(x_1, x_2, x_3) = 0 ; \quad f_2(x_1, x_2, x_3) = 0 ; \quad f_3(x_1, x_2, x_3) = 0 \]  \hspace{1cm} (14.50)

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, \(x_1 = x_1^{(0)} + \delta x_1; \quad x_2 = x_2^{(0)} + \delta x_2; \quad x_3 = x_3^{(0)} + \delta x_3\)  \hspace{1cm} (14.51)

and demand that the “corrections” satisfy the equations

\[
0 = f_1\left(x_1^{(0)} + \delta x_1, x_2^{(0)} + \delta x_2, x_3^{(0)} + \delta x_3\right) \\
0 = f_2\left(x_1^{(0)} + \delta x_1, x_2^{(0)} + \delta x_2, x_3^{(0)} + \delta x_3\right) \\
0 = f_3\left(x_1^{(0)} + \delta x_1, x_2^{(0)} + \delta x_2, x_3^{(0)} + \delta x_3\right)
\]  \hspace{1cm} (14.52)

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

\[
0 = f_1 + \frac{\partial f_1}{\partial x_1} \delta x_1 + \frac{\partial f_1}{\partial x_2} \delta x_2 + \frac{\partial f_1}{\partial x_3} \delta x_3 \\
0 = f_2 + \frac{\partial f_2}{\partial x_1} \delta x_1 + \frac{\partial f_2}{\partial x_2} \delta x_2 + \frac{\partial f_2}{\partial x_3} \delta x_3 \\
0 = f_3 + \frac{\partial f_3}{\partial x_1} \delta x_1 + \frac{\partial f_3}{\partial x_2} \delta x_2 + \frac{\partial f_3}{\partial x_3} \delta x_3
\]  \hspace{1cm} (14.53)

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.

**MAPLE:** Within MAPLE, solving systems of nonlinear equations is accomplished with the same commands—`solve` and `fsolve`—that we have used for solving single equations symbolically and numerically, though `solve` is more likely to fail when the equations to be solved are nonlinear. Essentially, for systems of equations, the first argument to `solve` or `fsolve` becomes a set containing the equations to be solved and the second argument becomes a set containing the variables for which solutions are sought.
IDL: For numerical solution of systems of nonlinear equations, the most commonly used IDL commands are `newton`, which uses an embellishment of Newton’s method that has improved convergence, and `broyden`, which implements a multidimensional secant method described for one variable in Section 14.6.3.

NUMERICAL RECIPES in FORTRAN: For numerical solution of systems of nonlinear equations, the primary FORTRAN subroutines in the Numerical Recipes library are `mnewt.f`, which implements a simple Newton’s method; `newt.f`, which adopts an embellished Newton’s method with improved convergence; and `broydn.f`, which implements a multidimensional secant method described for one variable in Section 14.6.3.

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. 14.15, 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

$$\frac{R(\theta, \alpha)}{\frac{v_0^2}{g}} = \frac{\cos \theta (1 - e^{-\tau})}{\alpha} = \frac{\tau \cos \theta}{\alpha(1 + \alpha \sin \theta)}$$
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. \textit{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 \textit{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 \( \beta = m'/m \). \textit{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

\[
\begin{align*}
m \frac{d^2 x_1}{dt^2} &= -kx_1 + k(x_2 - x_1) \\
m' \frac{d^2 x_2}{dt^2} &= -k(x_2 - x_1) + k(x_3 - x_2) \\
m \frac{d^2 x_3}{dt^2} &= -k(x_3 - x_2) - kx_3
\end{align*}
\]

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 $\sqrt{2} = 1.41421$. (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

\begin{align*}
L_0(x) &= 1 \\
L_1(x) &= x \\
L_2(x) &= \frac{1}{2}(3x^2 - 1) \\
L_3(x) &= \frac{1}{2}(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)
\end{align*}

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

$$\tan \alpha = \pm \tanh \alpha$$

Find the lowest half dozen natural frequencies for this bar. [A detailed discussion of this exercise can be found in N. H. Fletcher and T. D. Rossing, The Physics of Musical Instruments, Second Edition (Springer-Verlag, New York, 1991), pp. 57ff.]

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

$$\cot \alpha = \pm \tanh \alpha$$

Find the lowest half dozen natural frequencies for this bar. [A detailed discussion of this exercise can be found in N. H. Fletcher and T. D. Rossing, The Physics of Musical Instruments, Second Edition (Springer-Verlag, New York, 1991), pp. 57ff.]

14.16. The natural frequencies of the air in a spherical cavity are determined from the roots of the function $dj_n(x)/dx$, where $j_n(x)$ is the $n$-th order spherical Bessel function, the first three of which are

\begin{align*}
\sin \frac{x}{x}; \quad j_1(x) &= \frac{\sin x}{x^2} - \cos x \\
j_2(x) &= \left(\frac{3}{x^3} - \frac{1}{x}\right) \sin x - \frac{3}{x^2} \cos x
\end{align*}

Obtain graphs of these three functions versus $x$, find the lowest several roots of $j_0(x)$, $j_1(x)$, and $j_2(x)$, and find also the lowest several roots of $dj_0(x)/dx$, $dj_1(x)/dx$, and $dj_2(x)/dx$.

14.17. 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.
CHAPTER 14. FINDING ROOTS

14.18. The intensity \( I(x) \) in the diffraction pattern produced by a single slit is given by

\[
\frac{I(x)}{I_0} = \frac{\sin^2 x}{x^2}
\]

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 zeroes 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\(^{35}\) are 2.404825577, 5.5200781103, 8.6537279129, 11.7915344391, 14.9309177086, 18.0710639679. \textit{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

\[
V(x) = -\frac{V_0a^2(a^2 + x^2)}{8a^2 + x^4}
\]

where \( \xi = x/a \). Using at least three different methods and at least two different computational tools, find the coordinates \( \xi \) 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 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.

14.25. Write and test a FORTRAN program paralleling bisect.f but using Newton’s method to find the roots of $f(x)$. Your program, which you might call newton.f, should

- use the functions FUNC and 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.

14.26. Write and test a C program paralleling bisect.c but using Newton’s method to find the roots of $f(x)$. Your program, which you might call newton.c, should

- use the functions FUNC and 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 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 is assuredly built in to your computational tool (see the appropriate manuals), find the angle $\theta$ at which the first zero in the diffraction pattern lies, expressing that result as a multiple of $\lambda/d$. The resulting angle expresses the angular separation of two nearby objects such that the maximum in the diffraction pattern produced by one lies on top of the first minimum away from the maximum of the other. That separation is universally taken as a measure of the resolution of the optical system creating the images, and the condition requiring this positioning of the two maxima is called the Rayleigh criterion. Hint: The angle will be small, so you can safely use the approximation $\sin \theta \approx \theta$.

14.28. The Lennard-Jones potential energy $V_{LJ}$, which is given in terms of the coordinate $r$ by the expression

$$V_{LJ} = 4\epsilon \left[ \frac{\sigma^{12}}{r^{12}} - \frac{\sigma^6}{r^6} \right]$$

or

$$V_{LJ} = 4\epsilon \left[ \frac{1}{r^{12}} - \frac{1}{r^6} \right]$$
CHAPTER 14. FINDING ROOTS

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.

14.29. Study the demonstration programs xrtbis.f and xrtnewt.f, 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 $c$.

14.30. 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 $c$.

14.31. Read the Numerical Recipes books about the recipe rtsafe.f and the example program xrtsafe.f. Then recast the program sinrtbis.f (or sinrtnewt.f) to use rtsafe.f and test your program with a couple of functions of your choice.

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 $\theta = 39^\circ$, $40^\circ$, and $41^\circ$ are 0.7251744, 0.7259484, 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. 14.16, 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 $R_{\text{eff}} = \frac{V}{I}$ 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 $\$\text{HEAD}/data/freefall.dat$ 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 $a$,
CHAPTER 14. FINDING ROOTS

14.37. The file $\text{HEAD/data/freefall.dat}$ 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

$$\frac{\partial R}{\partial a} = 0 ; \quad \frac{\partial R}{\partial b} = 0 ; \quad \frac{\partial R}{\partial c} = 0$$

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 $\text{HEAD/data/freefall.dat}$ 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

$$md^2x/dt^2 = F(t) , \quad x(0) = 0 , \quad x(t_f) = a$$

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

$$
\begin{pmatrix}
? & ? & ? & \ldots & ? \\
? & ? & ? & \ldots & ? \\
? & ? & ? & \ldots & ? \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
? & ? & ? & \ldots & ? \\
\end{pmatrix}
\begin{pmatrix}
x_0 \\
x_1 \\
x_2 \\
\vdots \\
x_n \\
\end{pmatrix}
= 
\begin{pmatrix}
? \\
? \\
? \\
\vdots \\
? \\
\end{pmatrix}
$$

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 $n = 100$;
(c) \( t_f = 1 \, \text{s}, \, a = 10 \, \text{m}, \, F(t)/m = e^{-t} \, \text{N/kg} \) (exponentially decaying force; \( t \) in seconds), and \( n = 10 \); and

(d) same as (c) but with \( n = 100 \).

Verify that the exact analytic solution in the two cases is

(a) and (b): \( x(t) = 4t^2 + 6t \); (c) and (d): \( x(t) = e^{-t} - 1 + (11 - e)t \)

and compare the numerical results with the exact solution in each case. (To save you a hunt, remember that the numerical value of \( e \) is 2.71828459045.)

14.39. The electrostatic potential on the boundary of the square region in a plane shown in Fig. 14.17 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.,

\[
V_{12} = \frac{1}{4}(V_{11} + V_7 + V_{13} + V_{17}) \quad ; \quad V_{15} = \frac{1}{4}(V_{14} + V_{10} + 0.0 + V_{20})
\]

(a) Write out the twenty-five equations implied by this requirement, (b) cast them in the matrix form

\[
\begin{pmatrix}
\vdots & \vdots & \vdots & \ddots & \vdots \\
? & ? & ? & \cdots & ?
\end{pmatrix}
\begin{pmatrix}
V_1 \\
V_2 \\
V_3 \\
\vdots \\
V_{25}
\end{pmatrix}
= \begin{pmatrix}
? \\
? \\
? \\
\vdots \\
?
\end{pmatrix}
\]

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 \( 7 \times 7 \) 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., by a function of the form

\[
x(t) = Ae^{-bt} \cos(\omega t + \phi) + Ct + D
\]
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).
function lubisect, func, xlb, xub, tol=tol, itmax=itmax

if not keyword_set(tol) then tol=1.0e-6
if not keyword_set(itmax) then itmax=20

fl = call_function( func, xlb )
fu = call_function( func, xub )

if fl*fu gt 0.0 then begin
  print, 'The given interval does not contain a root.'
  return, xlb
endif

i = 0
repeat begin
  xm = 0.5*(xlb + xub)
  fm = call_function( func, xm )
  if fl*fm ge 0 then xlb = xm else xub = xm
  i = i + 1
endrep until (xub-xlb le tol) or (fm eq 0.0) or (i ge itmax)

return, xm
end
14.B Listing of bisect.f

PROGRAM BISECT

TOL = 0.0001

! ***** READ BRACKET VALUES *****
WRITE(*, '(1X, A)') 'Lower bound ='
READ(*,*) XA
WRITE(*, '(1X, A)') 'Upper bound ='
READ(*,*) XB

! ***** EVALUATE FUNCTION AT EXTREMES *****
FA = FUNC(XA)
FB = FUNC(XB)
IF ( FA*FB .GE. 0.0 ) THEN
   WRITE(*,'(1X,A)') 'No root in interval'
   GOTO 99
ENDIF
FM = FA

! ***** HUNT FOR ROOT *****
DO WHILE ( (XB-XA) .GE. TOL .AND. (FM .NE. 0.0) )
   XM = 0.5*(XB+XA)
   FM = FUNC(XM)
   IF (FA*FM .GE. 0.0) THEN
      XA = XM
   ELSE
      XB = XM
   ENDIF
ENDDO

! ***** OUTPUT RESULTS *****
WRITE(*, '(1X, A, E15.6)') 'ROOT = ', XM
WRITE(*, '(1X, A, E15.4)') 'XB-XA = ', XB-XA
WRITE(*, '(1X, A, E15.6)') 'FUNCTION = ', FM
99 END

! ***** DEFINE FUNCTION *****
FUNCTION FUNC(X)
   FUNC = X**3/10000.0 + X**2/200.0 - X/500.0 - 0.5
END
14.C 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( "Lower bound = " ); scanf( "%f", &xa );
    printf( "Upper 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 ) );
}
Appendix A

Introduction to $\LaTeX$

Note: All $\LaTeX$ source files (.tex, .template), all Windows batch files (.bat) program (.pro) files, and all UNIX command files (no file type) referred to in this chapter are available in the directory $\$HEAD/tex$, where (as defined in the Local Guide) $\$HEAD must be replaced by the appropriate path for your site.

During the 1970s, when Donald Knuth (a Stanford University computer scientist) was creating his monumental, several-volume series of books on all aspects of computer science, he recognized the need for a computer-based type-setting/word-processing system tailored to the needs of authors of technical manuscripts. Interrupting his main project, he developed $\TeX$ as the essential engine to support computerized technical type setting. The first version of $\TeX$ was made available to the using public in 1978 and a revised (and final) version was published in 1984. Written by Leslie Lamport, $\LaTeX$ is a versatile and extensive collection of macros that sit on top of $\TeX$ and facilitate exploitation of $\TeX$'s capabilities. Broadly, $\LaTeX$ is a document preparation system that formats equations, tables, and illustrations as easily as plain text. That the whole complex ($\TeX$, $\LaTeX$, and many other components) has been deliberately placed in the public domain,\(^3\) that carefully tested versions exist for essentially every computing platform and operating system, and that many scientific journals and an increasing number of publishers will accept manuscripts submitted as $\LaTeX$ files together provide substantial incentive for learning to use this tool. Although it is not wysiwyg,\(^4\) it has powerful and sophisticated capabilities, most of which are not described in this Appendix. We convey only the basics with, however, the hope that we will cause you to realize that essentially any desired formatting at all should be possible if only you can figure out how to achieve it. For further information, try surfing the web in your favorite browser and looking at the $\LaTeX$ User’s Guide and Reference Manual (which we will refer to as The $\LaTeX$ Manual), the $\LaTeX$ Companion, and/or to any of the several other $\TeX$ and $\LaTeX$ books.\(^5\) The wisdom of taking fifteen minutes every now and then to browse in these publications cannot be overstressed; as you become more skilled, the fine print and other subtleties will gradually be more meaningful.\(^6\)

\(^1\) As per Knuth’s instruction (see the first two paragraphs of Chapter 1 of The $\TeX$book as identified in Section A.18), $\TeX$ is pronounced ‘tech’ as in ‘technology’.

\(^2\) Here, individuals disagree on the pronunciation of the ‘la’, and Leslie Lamport offers no guidance. Some say lah$\TeX$ while others say lay$\TeX$, with the emphasis on the first syllable in both cases. We simply must become accustomed to each other’s preferences.

\(^3\) 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 $\TeX$/$\LaTeX$ system can be downloaded.

\(^4\) what you see is what you get.

\(^5\) See Section A.18 for more detailed references.

\(^6\) You may also find the web pages of TUG at the URL http://www.tug.org to be valuable. One link on that page points to an engaging description of the history of the development of $\TeX$. 

457
A.1 Creating a Simple Document

Producing a final document with \LaTeX involves a number of steps, beginning with the creation of an ASCII text file containing the \LaTeX “source code”—hereafter simply “code”—for the document. Appropriate conversion programs are then invoked to create either a PostScript or a PDF file containing the formatted document.\footnote{Other formats (HTML, ePUB, ...) may also be produced, though this Appendix will not address those alternatives.} Finally, the PostScript or PDF file will be viewed on the screen or sent to a printer. In this section, we describe how to structure the code for a \textit{very} simple document and then how to carry out the remaining steps to convert that code into a displayed or printed document. The remainder of this Appendix will explain numerous embellishments that might be invoked in the code. Processing that code to produce the final document is (more or less) independent of the complexity of the code.

To keep the discussion comparatively simple, we assume initially that your document incorporates no figures, does not have internal cross references, and includes neither a table of contents or an index. Those embellishments will be described in later sections.

A.1.1 Structuring a \LaTeX Source File

As input, \LaTeX requires an ASCII file, which can be created with any text editor (\texttt{xemacs, notepad, wordpad, winedt, gedit, ...}) and which contains both the text of the desired document and embedded formatting commands. All of \LaTeX’s commands \textit{begin} with a backslash (\texttt{	extbackslash{}}). \LaTeX distinguishes upper and lower case letters; while most standard commands and parameters use exclusively lower case letters, we must nonetheless pay attention to case.

Three commands are sufficient to create the simplest document containing straight text formatted with \LaTeX’s defaults. The very first required command in the code for \textit{any} \LaTeX document specifies the document \textit{class} and has the form\footnote{Be aware that some of the features described in this Appendix are specific to \LaTeX2\textepsilon and will not work with the previous version (\LaTeX 2.09). \LaTeX2\textepsilon, however, will automatically enter 2.09 emulation mode if the first line in the code is the now obsolete command \texttt{\documentstyle} instead of the new command \texttt{\documentclass}.}

\begin{verbatim}
\documentclass{Class}  \texttt{or \ \documentclass[options]{Class}}
\end{verbatim}

where

- \textit{Class} is any one of the options shown in Table A.1 and
- \textit{options} may be omitted altogether if the defaults are acceptable. The most commonly used options, one or more of which may be separated by commas in a string containing \texttt{no} spaces, are listed in Table A.2.

The document itself must be enclosed between the second and third of the essential commands, the second marking the beginning of the document proper and having the form

\begin{verbatim}
\begin{document}
\end{verbatim}

and the third marking the end of the document and having the form

\begin{verbatim}
\end{document}
\end{verbatim}
A.1. CREATING A SIMPLE DOCUMENT

Table A.1: Document classes valid in standard \LaTeX. Other available classes may be described in the Local Guide.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>article</td>
<td>This class is by far the most common. It is used for most short documents. See Sections 2.2.2 and C.5.1 in The \LaTeX Manual.</td>
</tr>
<tr>
<td>report</td>
<td>This class is generally used for longer documents. See Sections 2.2.2 and C.5.1 in The \LaTeX Manual.</td>
</tr>
<tr>
<td>book</td>
<td>This class is meant for actual books. See Sections 5.1 and C.5.1 in The \LaTeX Manual.</td>
</tr>
<tr>
<td>slides</td>
<td>This class can be used in creating originals from which overhead transparencies to be used in presentations can be made. The type size, including the size used for mathematical symbols, is quite large, and transparencies readable from some distance will be produced. See Sections 5.2 and C.5.1 in The \LaTeX Manual.</td>
</tr>
<tr>
<td>letter</td>
<td>This class facilitates creating letters. See Sections 5.3 and C.5.1 in The \LaTeX Manual.</td>
</tr>
</tbody>
</table>

Table A.2: The most common options for the \texttt{documentclass} command. Numerous other options are described in Section C.5.1 in The \LaTeX Manual and in the other references listed in Section A.18.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>10pt or 11pt or 12pt</td>
<td>(default = 10pt) Specifies default type size.</td>
</tr>
<tr>
<td>letterpaper or legalpaper or a4paper or ...</td>
<td>(default depends on site but is usually letterpaper in the US) Specifies size of paper to be used.</td>
</tr>
<tr>
<td>landscape</td>
<td>(default is portrait orientation) Specifies that text is to be formatted for landscape orientation on the specified paper size.</td>
</tr>
<tr>
<td>oneside or twoside</td>
<td>(default = oneside) Specifies whether output is to be formatted for single- or double-sided printing. The twoside option allows for left and right margins and running head positions to be different on odd and even numbered pages.</td>
</tr>
<tr>
<td>onecolumn or twocolumn</td>
<td>(default = onecolumn) Specifies one- or two-column formatting on each page.</td>
</tr>
</tbody>
</table>

While there can be material in the code beyond this final command, none of that material will be read or processed by \LaTeX.

The three commands described in the previous paragraph are mandatory in all documents. Supplemented by the additional feature that a blank line in the code triggers a new paragraph, they are also sufficient for the creation of code for any document consisting of nothing but paragraphs of straight text to be formatted in accordance with all of \LaTeX’s defaults. Table A.3 shows a listing of a simple code that invokes only these commands.

Whether the code is simple (as in Table A.3) or much more elaborate, producing the final printed document involves additional steps. One first creates a printable file—PostScript and PDF are the most common—containing the formatted document and then displays that file on a screen and/or sends it to a printer.
Table A.3: A simple \LaTeX{} code.

\documentclass{article}
\begin{document}

This sample illustrates the simplest code containing the mandatory commands and a brief text. Since no optional formatting commands have been included, the text will be formatted using all of the built-in defaults (margins, paragraph indent, type size and font, etc.).

The blank line preceding this line in the code will trigger a new paragraph. Each paragraph is indented, but note that there is no extra space between paragraphs. Note also that the lines in the code can be quite ragged; they will be filled and justified in the processing that produces the final copy.

\end{document}

A.1.2 Creating a PostScript File

Converting the \LaTeX{} source file into a PostScript file describing the desired document involves two steps:

1. First, we must “compile” the code by processing the file with \LaTeX{}, a task accomplished either (1) by typing a command like

latex filename  

(\text{Default extension for the input file is .tex.})

where \texttt{filename} specifies the input file to be processed, or (2) by selecting an item from a menu in a graphical user interface (GUI).\footnote{See the \textit{Local Guide} for the precise command at your site.} However \LaTeX{} is invoked, it will create several output files, all of which will have the same name as the input file except for the extension: (1) a binary output file, with extension .dvi, which contains the translation of the original file into \TeX{}’s \textit{device} independent language; (2) an ASCII log file, with extension .log, which contains an expanded log of the messages that appear on the screen as \texttt{latex} works its magic on the input file; and (3) an auxiliary ASCII file, with extension .aux, which contains information that is important only if the code makes use of \LaTeX{}’s capabilities for generating internal cross references.\footnote{These capabilities will be described in later sections of this Appendix. For now, we may ignore messages relating to the .aux file, noting only that, when we ultimately do make use of internal references, the \textit{first} pass of the document through \LaTeX{} writes the .aux file and a \textit{second}—and occasionally a third—pass is necessary to incorporate that information in the final formatting of the pages.}

The resulting .dvi file can be directly displayed on the screen (see Section A.1.4).\footnote{Remember that we have limited the present discussion to documents containing no figures. See Section A.7 for the adjustments to be made if PostScript or PDF figures are included.}

2. Second, we must translate the .dvi file into a file that can be understood by the printer on which the document is to be printed. Most often, the .dvi file will be translated into...
A.1. CREATING A SIMPLE DOCUMENT

Table A.4: The output from the code in Table A.3. The page number at the bottom of the page has been cut off in this display.

This sample illustrates the simplest code containing the mandatory commands and a brief text. Since no optional formatting commands have been included, the text will be formatted using all of the built-in defaults (margins, paragraph indent, type size and font, etc.).

The blank line preceding this line in the code will trigger a new paragraph. Each paragraph is indented, but note that there is no extra space between paragraphs. Note also that the lines in the code can be quite ragged; they will be filled and justified in the processing that produces the final copy.

PostScript, the most common program for effecting that translation being \texttt{dvips}. The necessary translation will be achieved either (1) by typing a command like

\begin{verbatim}
dvips -o filename.ps -t letter filename
\end{verbatim}

which translates the entire file into PostScript, stores it in a file, and (with the \texttt{-o} option here used) gives the stored file the same name as the input file but with extension \texttt{.ps} and with the \texttt{-t} option forces use of letter size paper or (2) by selecting an item from a menu.

To be more explicit, the file \texttt{texsample1.tex}, contains the \LaTeX code in Table A.3. Once this file has been copied to the default directory, we would produce the \texttt{.dvi} and \texttt{.ps} files with statements like

\begin{verbatim}
l latex texsample1
dvips -o texsample1.ps -t letter texsample1
\end{verbatim}

or equivalent selections from a menu.

\section*{A.1.3 Creating a PDF File}

PDF files can be created from the \LaTeX code in at least two ways:

1. Starting with \LaTeX source file created in Section A.1.1, one submits to the operating system the single statement

\begin{verbatim}
pdflatex filename
\end{verbatim}

or, for the sample above,

\begin{verbatim}
pdflatex texsample1
\end{verbatim}

\footnote{Additional information about \texttt{dvips} can be found by typing the command \texttt{dvips} with no arguments. Further, the option \texttt{-pp} allows selection of specific pages from the full document, the options \texttt{-p} (first page) and \texttt{-l} (last page) allow selection of a range of pages from the full document, and the option \texttt{-t} allows printing of the output in landscape orientation rather than in the default portrait orientation. Even more detailed information is available in the on-line help for \texttt{dvips}, accessed in many systems by typing the statement \texttt{texdoc dvips} at a \texttt{Shell} prompt or, in the UNIX and LINUX operating systems, the statement \texttt{man dvips} at a \texttt{Shell} prompt.}

\footnote{If the explicit specification of the output file is omitted, \texttt{dvips} will attempt to send the file directly to the printer—and the whole operation will probably fail. Further, the PostScript file will then not be stored for subsequent use.}

\footnote{This option is often the default. The option \texttt{-t landscape} will override the default portrait orientation.}

\footnote{Check the \texttt{Local Guide} for more details. In particular, some sites may have implemented the shorthand command \texttt{dvip filename} for the command \texttt{dvips -o filename.ps filename}.}

\footnote{The program \texttt{pdflatex} is normally installed automatically whenever \LaTeX is installed.}

\footnote{This option is often the default. The option \texttt{-t landscape} will override the default portrait orientation.}
which will produce directly a PDF file named filename.pdf or texsample1.pdf as well as two additional files, specifically the .aux (which we can ignore for now) and the .log file containing an expanded log of the messages that appear on the screen as latex works its magic.

2. Starting with the .ps file produced in Section A.1.2, one executes the single statement

\[ \text{ps2pdf filename.ps} \]  
(No default file type)

or, for the sample above

\[ \text{ps2pdf texsample1.ps} \]

which will produce a PDF file named filename.pdf or texsample1.pdf from the .ps file.

Exploration of other routes to a PDF file is left to the reader. In particular, the program dvipdfm can sometimes be used to convert the .dvi file created above directly to a PDF file without creating the PostScript file as an intermediary.

A.1.4 Displaying the Document on the Screen

Numerous programs for displaying documents produced with \LaTeX on the screen exist. Among the more common are the following:

- (for displaying a .dvi file) \texttt{xdvi} for UNIX and LINUX platforms and \texttt{yap} (yet another previewer) for windows platforms. These programs are described more fully in Section A.14.
- (for displaying a PostScript file) \texttt{ghostview}, which is available for almost all platforms.
- (for displaying a PDF file) Adobe \texttt{acroread}, which is available for almost all platforms.

These programs are invoked by double-clicking ML on an icon for the program and then opening the desired file, by double-clicking ML on an icon for the file to be displayed, by executing an appropriate statement as a command in a \textit{Shell} window, or by double-clicking ML on the name of the file in a directory displayed on the screen.\footnote{See the Local Guide for specifics at your site.}

A.1.5 Printing the Document

Programs (\texttt{xdvi}, \texttt{yap}, \texttt{ghostview}, \texttt{acroread}) for displaying a file on the screen often have an item in the File menu to request printing of the file on an available printer, and many times the print utility thereby invoked offers options for double-sided printing, scaling of the output, ... Often, a simple command like

\begin{verbatim}
UNIX/LINUX/MAC       Windows
lp filename.ps       print filename.ps
\end{verbatim}

submitted from a \textit{Shell} or \textit{Command} window will request a printed copy of the file, though the destination printer must be known to the operating system and able to translate PostScript.\footnote{Printers without this capability are rare.} In the case of the typed command, the extension must this time be explicitly present, since the programs \texttt{lp} and \texttt{print} make no assumptions about file type.\footnote{Again, check the Local Guide for details on how to print a file.} In any case, the resulting output for our sample file is shown in Table A.4.
A.2 Specification of Global Style: The Preamble

The standard document classes have default settings for the page setup (area allocated to text, margins, paragraph indent, paragraph separation, line spacing, etc.). Only occasionally are these defaults actually appropriate for the document being prepared. Therefore, it is frequently necessary to modify the global style of the document. Commands that accomplish this end are normally placed in the preamble—the section between the command `\documentclass` and the command `\begin{document}`—and affect the entire document. The most commonly used commands in the preamble invoke the general command

```
\setlength{LengthParameter}{Value}
```

to set the length parameter identified by `LengthParameter` to the value specified by `Value`. A length can be a positive or negative value given in points\(^{20}\) (abbreviated `pt`), centimeters (abbreviated `cm`), or inches (abbreviated `in` or, when we wish to make the dimension immune to global changes of scale, `truein`). Further, to give \LaTeX{} some flexibility in placing text on the page, `rubber` values can be specified. For example, a value of `9pt plus 1pt minus 2pt` gives \LaTeX{} authority to “cheat” on the value over the range from 7 points to 10 points. The most useful length parameters are identified in Table A.5. Thus, the commands

```
\setlength{\textheight}{9truein}
\setlength{\textwidth}{6truein}
\setlength{\oddsidemargin}{0.25truein}
\setlength{\topmargin}{-0.5truein}
\setlength{\parindent}{20pt}
\setlength{\parskip}{6pt plus 2pt minus 1pt}
```

placed in the preamble will change the defaults to specify a 6" × 9" area of text centered on an 8.5" × 11" page with a 20 point paragraph indent and an extra 5–8 points between paragraphs. Indeed, these are (almost\(^{21}\)) the settings used for this book.

A few issues of global style are specified by an optional argument in the command `\documentclass`. The most common of these arguments specifies the type size. By default, the document will be set in 10 point type, which is standard for the main text in many journals. To change the size of the characters throughout a document, modify the `\documentclass` command to\(^{22}\)

```
\documentclass[TypeSize]{Class}
```

where, in the `article`, `book`, and `report` classes, `TypeSize` can be one of `10pt` (the default), `11pt`, or `12pt`. The `slides` class admits these specifications of size, but ignores them. The main text in this book is set in \LaTeX{}'s 10 point type, which is also common in many technical journals.

Several other global specifications can be included in the preamble or as optional arguments in the command `\documentclass`, including specifications regarding positioning and style of page headers, definitions of new commands, more subtle changes in the global style, selection of two-column format, specifications to anticipate ultimate printing on both sides of the page, etc. For details, the reader is referred to *The \LaTeX{} Manual*.

Specific instructions for document styles can be saved in files for easy incorporation in documents as they are created. Several possibly useful templates are enumerated in Table A.6. Consult the

\(^{20}\)The point, which is a standard printer’s measure of length, is 1/72" (72 points per inch).

\(^{21}\)The difference lies in the specification of the left margins. For this book (which uses double-sided printing), \LaTeX{}’s ability to specify one left margin (with `\oddsidemargin`) for odd-numbered pages and a different left margin (with `\evensidemargin`) for even-numbered pages has been exploited to keep the text more fully out of the binding than would otherwise be the case.

\(^{22}\)By convention in \LaTeX{}, optional arguments to a command are enclosed in square brackets and mandatory arguments are enclosed in curly braces.
Table A.5: Selected parameters that affect the size and positioning of the text area on a page. The
given default values apply to the \texttt{article} style. Further information can be found in Section 6.4.1,
at the very end of Section C.5.3, and in Fig. C.3 in \textit{The \LaTeX\ Manual}.

\setlength{\textwidth}{6in}  \setlength{\textheight}{9in}  \setlength{\oddsidemargin}{0.25in}  \setlength{\evensidemargin}{0.25in}  \setlength{\topmargin}{0.25in}  \setlength{\parindent}{15.0pt}  \setlength{\parskip}{0.0pt}

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textwidth</td>
<td>4.75&quot;</td>
<td>Specifies the horizontal dimension of the page occupied by text. Its value is commonly 6&quot; for an 8.5&quot;×11&quot; page.</td>
</tr>
<tr>
<td>\textheight</td>
<td>7.375&quot;</td>
<td>Specifies the vertical dimension of the page occupied by text. Its value is commonly 9&quot; for an 8.5&quot;×11&quot; page.</td>
</tr>
<tr>
<td>\oddsidemargin</td>
<td>0.75&quot;</td>
<td>For all pages (single-sided output) or odd-numbered pages (double-sided output), specifies the left margin—distance from the left edge of the page to the left edge of the region occupied by text—to be 1&quot; plus the specified value. The value 0.25&quot; will center 6&quot; wide text on an 8.5&quot;×11&quot; page.</td>
</tr>
<tr>
<td>\evensidemargin</td>
<td>0.75&quot;</td>
<td>Only for even-numbered pages with double-sided output, specifies the left margin—distance from the left edge of the page to the left edge of the region occupied by text—to be 1&quot; plus the specified value. The value 0.25&quot; will center 6&quot; wide text on an 8.5&quot;×11&quot; page.</td>
</tr>
<tr>
<td>\topmargin</td>
<td>0.25&quot;</td>
<td>Specifies the top margin, i.e., the distance from the top edge of the page to the top edge of the header above the main region occupied by text, to be 1.0&quot; plus the specified value. By default, the height of the header (specified by \texttt{\headheight}) and the separation of the header from the main text (specified by \texttt{\headsep}) together add to 0.5&quot;, so—if the defaults for \texttt{\headheight} and \texttt{\headsep} are accepted—we can pretend that \texttt{\topmargin} specifies the distance from the top edge of the page to the top edge of “real” text to be 1.5&quot; plus the specified value. Thus, for example, the value −0.5&quot; will center 9&quot; high text on an 8.5&quot;×11&quot; page and place the header line—if any—0.5&quot; above the first line of text.</td>
</tr>
<tr>
<td>\parindent</td>
<td>15.0 points</td>
<td>Specifies the paragraph indent.</td>
</tr>
<tr>
<td>\parskip</td>
<td>0.0 points</td>
<td>Specifies the extra space between paragraphs.</td>
</tr>
</tbody>
</table>

Table A.6: Templates in the directory \texttt{$\HEAD/tex$}.

<table>
<thead>
<tr>
<th>Template</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{lu_article.template}</td>
<td>Specifies a general article style, including a 6&quot;×9&quot; text area centered on an 8.5&quot;×11&quot; page.</td>
</tr>
<tr>
<td>\texttt{lu_cpl.template}</td>
<td>Specifies the style of CPL publications.</td>
</tr>
<tr>
<td>\texttt{lu_sloan.template}</td>
<td>Specifies the two-column style used for the proceedings of the Sloan/Lawrence conference, including capacity for a two-column wide title and abstract on the first page.</td>
</tr>
<tr>
<td>\texttt{lu_letter.template}</td>
<td>Supplies a starting format for business letters.</td>
</tr>
<tr>
<td>\texttt{lu_memo.template}</td>
<td>Supplies a starting format for memos.</td>
</tr>
</tbody>
</table>
A.3. IN-TEXT SPECIFICATION OF LOCAL STYLE

Table A.7: Commands for changing type style. Note: Some of these commands may exhibit unexpected behaviors when used in math mode and one of them (\sc) is invalid in math mode.

<table>
<thead>
<tr>
<th>Language</th>
<th>Command</th>
<th>Function</th>
<th>Sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>\TeX</td>
<td>{\rm text}</td>
<td>Sets text in medium, Roman, upright type.</td>
<td>Sample</td>
</tr>
<tr>
<td>\TeX</td>
<td>{\sl text}</td>
<td>Sets text in medium, Roman, slanted type.</td>
<td>Sample</td>
</tr>
<tr>
<td>\TeX</td>
<td>{\it text}</td>
<td>Sets text in medium, Roman, italic type.</td>
<td>Sample</td>
</tr>
<tr>
<td>\TeX</td>
<td>{\tt text}</td>
<td>Sets text in medium, typewriter, upright type.</td>
<td>Sample</td>
</tr>
<tr>
<td>\TeX</td>
<td>{\bf text}</td>
<td>Sets text in bold, Roman, upright type.</td>
<td>Sample</td>
</tr>
<tr>
<td>\TeX</td>
<td>{\sc text}</td>
<td>Sets text in medium, Roman, small caps type.</td>
<td>SAMPLE</td>
</tr>
<tr>
<td>\LaTeX</td>
<td>{\sf text}</td>
<td>Sets text in medium, sans serif, upright type.</td>
<td>Sample</td>
</tr>
<tr>
<td>\LaTeX</td>
<td>{\em text}</td>
<td>Sets text in emphasized type.</td>
<td>(See text.)</td>
</tr>
<tr>
<td>\LaTeX</td>
<td>\texttt{\textbf{\textit{\textsc{\emph{}}}}}</td>
<td>Sets text in emphasized type.</td>
<td>(See text.)</td>
</tr>
</tbody>
</table>

Local Guide for information about possible additional templates available at your site.

A.3 In-Text Specification of Local Style

In addition to the global issues discussed in Section A.2, documents are affected locally by many less extensive changes of style. We have already mentioned the use of a blank line to trigger a new paragraph. Commands for accomplishing other common local changes are described briefly in this section.

A.3.1 Type Style

In the terminology of type setters, the phrase type style refers collectively to combinations of three independent characteristics of the type. In the most sophisticated description, type style is specified by selecting the series (medium, bold), family (Roman, sans serif, typewriter), and shape (upright, italic, slanted, small cap) of the desired style.\(^{23}\) Each characteristic is specified independently of the other two. Repeated selection of all three characteristics, however, is cumbersome, so—as enumerated in Table A.7—\LaTeX provides simpler ways to select the more common combinations. Some of these mechanisms make use of declarations,\(^{24}\) which change the type style until it is explicitly changed again. If, for example, we included the declaration \bf at some point in our code, everything in our document from that point until we change the style with another declaration would be set in bold, Roman, upright type. To limit the scope of this declaration to some portion of the text, the text to be “emboldened”, including the declaration, would be enclosed in curly braces (as shown in the table). Once \LaTeX’s processing has passed beyond the point of the closing curly brace, the declaration is no longer in effect and the type style reverts to what it was before the opening curly brace.

Emphasis of a word or a phrase now and then would, in normal use, be achieved with italic type. In \LaTeX, the declaration \it might be used. The declaration \em, however, is preferred because it is sensitive to the current type style. When the current type style is Roman, \em will shift to italic type; when the current type style is italic, \em will shift to Roman type. Thus, \em can be used inside a phrase that is already being emphasized. Emphasized text inside of emphasized text that is itself embedded in Roman text will be set in Roman text to contrast with the italic style of the text that immediately surrounds it.

\(^{23}\)The default is medium series, Roman family, upright shape—the type style used for this book.

\(^{24}\)Because it is sometimes difficult to remember which “commands” are commands and which are actually declarations, we index both as commands.
Note also in Table A.7 that emphasized text can be achieved not only with the *declaration* \emph{ but also with the *command* \emph{. The former form is perhaps preferable for longer phrases; the latter would be used for a word or two. Each achieves the same effect.

To be especially fastidious, we should recognize that return from italic or emphasized type to other type styles may result in too little space between the last italic or emphasized letter and the first Roman (say) letter. The command / inserted immediately after the last italicized or emphasized letter instructs \LaTeX{} to insert the last letter’s *italic correction*; use of this correction will usually improve the legibility of the final copy. Thus, the careful way to specify an emphasized word is, for example, \{\em and\}. (The italic correction is not necessary in all situations, e.g., if the following character is a period or a comma, which explains why it cannot be automatically and always inserted.)

In using one or another of these simple specifications, we are accepting particular combinations of series, family, and shape. \LaTeX{} provides two ways to take advantage of the full flexibility with *separate* specification of these three characteristics. For example, we might use declarations to specify bold, sans serif, upright type with the construction

```
{\bfseries\sffamily\upshape text}
```

Alternatively, we might use the commands

```
\textbf{ \textsf{ \textup{ text} } }
```

to achieve the same end with a shorter phrase. The full set of available declarations and commands includes

```
\mdseries, \bfseries, \textmd{...}, \textbf{...}
```

to specify medium and bold series, respectively,

```
\rmfamily, \sffamily, \ttfamily, \textrm{...}, \textsf{...}, \texttt{...}
```

to specify Roman, sans serif, and typewriter families, respectively, and

```
\upshape, \itshape, \slshape, \scshape ,
\textup{...}, \textit{...}, \textsl{...}, \textsc{...}
```

to specify upright, italic, slanted, and small-cap shapes. A quick return to the default specified in the originally invoked document class is accomplished with the declaration \normalfont or the command \textnormal.

### A.3.2 Type Size

Type size is independent of type style. The global type size for a particular document is specified in the command \documentclass as described in Section A.2. A local change in type size is accomplished by one or another of the declarations in Table A.8. The selected type size is determined relative to the global type size specified in the command \documentclass. Again curly braces are used to limit the scope of the change in size.

To combine a change in type *style* with a change in type *size*, place the specification of type size *first*, e.g., \{\Large\bf text\} to select large bold-face type.
Table A.8: Commands for changing type size. Note that the actual size produced by each command is relative to the global type size (10 pt, 11 pt, or 12 pt) in use. Note also that, in some classes, adjacent members in this sequence may be assigned to the same type size.

<table>
<thead>
<tr>
<th>Command</th>
<th>Function</th>
<th>Sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>\tiny text</td>
<td>Sets text in tiny type.</td>
<td>Sample</td>
</tr>
<tr>
<td>\scriptsize text</td>
<td>Sets text in scriptsize type.</td>
<td>Sample</td>
</tr>
<tr>
<td>\footnotesize text</td>
<td>Sets text in footnotesize type.</td>
<td>Sample</td>
</tr>
<tr>
<td>\small text</td>
<td>Sets text in small type.</td>
<td>Sample</td>
</tr>
<tr>
<td>\normalsize text</td>
<td>Sets text in normalsize type.</td>
<td>Sample</td>
</tr>
<tr>
<td>\large text</td>
<td>Sets text in large type.</td>
<td>Sample</td>
</tr>
<tr>
<td>\Large text</td>
<td>Sets text in Large type.</td>
<td>Sample</td>
</tr>
<tr>
<td>\Huge text</td>
<td>Sets text in huge type.</td>
<td>Sample</td>
</tr>
<tr>
<td>\Huge text</td>
<td>Sets text in Huge type.</td>
<td>Sample</td>
</tr>
</tbody>
</table>

A.3.3 White Space

Some aspects of the way the components (text, tables, graphs, ...) of a document are placed in the available space are controlled by the length parameters set in the preamble and discussed in Section A.2. Additional features of \LaTeX that give control over this feature of a document include

- The command \noindent to suppress paragraph indentation for a single paragraph.
- The commands newpage and clearpage to force a new page. The first of these commands terminates the current page and starts a new page. The second also flushes out any accumulated tables and figures that have not yet been output.
- The length parameter \baselineskip to specify the linespacing. This parameter is set with the command \setlength described in Section A.2. The default value is 12 points, which yields single spacing. The value 18 points specifies space and a half, and the value 24 points specifies double spacing. This specification must be placed after the command \begin{document}, and it can be changed whenever appropriate to the text. See, however, Section A.3.4 for a comment about limitations of this approach to setting the linespacing.
- The commands \quad and \qquad, which insert an en space and an em space, respectively. (An en space is about the width of the letter x, and an em space is about the width of the letter M, both in the current font.)
- The command \vspace{Value} to generate vertical space. Here, Value can be any defined length parameter, e.g., \vspace{\parskip}, or a specific length, e.g., \vspace{2.0truein}. If the space happens to occur at the top of a new page, it will be omitted, but the command \vspace*{Value} will force the space to be included even if it is at the top of a page.
- The command \hspace{Value} to generate horizontal space. Here, Value can be any defined length parameter, e.g., \hspace{\parindent}, or a specific length, e.g., \hspace{36.0pt}. If the space happens to occur at the end of a line, it will be omitted, but the command \hspace*{Value} will force the space to be included even if it is at the end of a line.
- The commands \settowidth, \settoheight, and \settodepth, which return lengths for the corresponding dimensions of the box that will enclose whatever is the argument of the
command. No printed output is produced by these commands. We might, for example, define a new length parameter \texttt{dblast} as the width of the box accommodating two asterisks with the two commands

\newlength{dblast}
\settowidth{dblast}{**}

and then insert the space for two asterisks (without actually displaying them) with the command

\hspace{dblast}

These commands are described in detail at the end of Sections 6.4.1 and C.13.1 in \textit{The \LaTeX\ Manual}.

\section*{A.3.4 Environments}

Sometimes, the formatting to be accomplished requires a more sophisticated definition of a change than is possible within the framework of a simple declaration or command with a few arguments. \LaTeX\’s \textit{environments} are more versatile and flexible than declarations and commands. For example, to present text in a narrower paragraph, as is conventionally done with extended quotations, we might invoke one of the constructions

\begin{quote}
Text of quotation.
\end{quote}

\begin{center}
First line \\
Second line \\
Third line \\
\ldots
\end{center}

Normally, the command \texttt{\begin{...} will be placed on a line by itself, the text may spread over several lines, and the command \texttt{\end{...} will be placed on a line by itself. Left and right margins are indented equally. Unless the text includes its own formatting specifications, however, the first construction will indent each new paragraph and will place \textit{no} extra space between paragraphs while the second will place extra space between paragraphs and will \textit{not} indent each new paragraph.

For a second example, if we wished to center a line or lines, we might invoke the similar construction

\begin{center}
First line \\
Second line \\
Third line \\
\ldots
\end{center}

Normally, the command \texttt{\begin{...} will be placed on a line by itself, the text may spread over several lines, each line in the final output will in the code be separated from its predecessor with the command \texttt{\\\}, and the command \texttt{\end{...} will be placed on a line by itself.

These two examples introduce the construction called an \textit{environment}, which in general has the format

\begin{EnvironmentName} \ldots \end{EnvironmentName}

We met the \texttt{document} environment early on. Beyond the \texttt{quotation}, \texttt{quote}, and \texttt{center} environments, the most commonly used environments include \texttt{verbatim}, \texttt{flushleft}, \texttt{flushright}, \texttt{itemize}, \texttt{enumerate}, \texttt{list}, \texttt{displaymath}, \texttt{equation}, \texttt{eqnarray}, \texttt{figure}, \texttt{table}, \texttt{tabular}, \texttt{minipage}, \texttt{picture}, and \texttt{thebibliography}. Environments can be thought of as mini-documents within a document. They have default settings, but many of them possess parameters that the user can modify to customize the details of the environment. Further, as described in Sections 3.4 and C.8 in \textit{The
individual users can supplement the standard set by defining additional environments suited to the circumstances of the document. Individual exercises in this book, for example, are formatted by a user-defined environment.

The length parameter \baselineskip interacts in interesting ways with the formatting of text contained within environments. Changing \baselineskip will affect only the spacing in the main text of the document, including text in itemize, enumerate, and verbatim environments. Footnotes, material in tabular environments, table and figure captions, and perhaps other components will remain single spaced unless—as described in Section C.3.2 in The \LaTeX Manual—the parameter \baselinestretch is also changed with the command \renewcommand. This additional change, however, has interesting side effects. (For example, the spacing of lines within footnotes will be changed but the spacing between separate footnotes on the same page remains single!) Tampering with global style can at times have unintended (and unwanted) consequences. Beware!

A.3.5 \LaTeX Packages

As a publishing system, \LaTeX is infinitely extendable. Rather than incorporate all manner of special tools within the basic program, its designers provided a means by which additional features could be added through the use of packages, the features of which can be made available in any specific code by placing the command

\usepackage{ PackageName }

in the preamble The \LaTeX Companion describes numerous packages, including amstex, babel, color, graphics, graphicx, graphpap, ifthen, latexsym, makeidx, verbatim, pict2e, and showidx. A brief description of many of these packages is included in Section C.5.2 of The \LaTeX Manual. The statement texdoc PackageName at a Shell prompt also will bring up documentation on several of these packages.

In addition, some programs, especially those with notebook capabilities, have the ability to write the contents of the notebooks into \LaTeX source files. Usually, those programs make use of their own special \LaTeX commands and supply program-specific \LaTeX packages which must be accessible, either at the proper point in the \LaTeX directory structure or in the directory from which you run \LaTeX when processing a \LaTeX file produced by the programs. Details on those additional packages will be found in the manuals provided by the vendors of the programs.

A.3.6 Miscellaneous Other Capabilities

Several additional capabilities merit particular comment:

- To force \LaTeX to keep words together on a single line and/or to prevent \LaTeX from inserting additional space as it justifies a line, use a tilde ~ instead of a space between the words. The tilde should also be used after a period that does not end a sentence. For example, we should type 'Mr.~Cook' rather than 'Mr.␣Cook' in our file so as to produce 'Mr. Cook' rather than 'Mr. Cook' in our output. In the second form, there is more space between the abbreviation and the name because \LaTeX automatically inserts extra space after each period (which is assumed to mark the end of a sentence). \LaTeX may add even more space in the second case as it justifies the line between the margins.

- To generate a dash, use -, --, or ---, depending on the length of the required dash. One hyphen produces a hyphen, as in two-column; two hyphens in a row produce a slightly longer

\cite{TheLaTeXManual} The symbol '_-' underscores the presence of a space at the indicated point; it is not a character explicitly present in the code.
dash (an en dash in printers’ terminology), conventionally used to indicate ranges, e.g., 10–12; three hyphens in a row produce a still longer line (—, an em dash in printers’ terminology), sometimes called a punctuation dash and conventionally used in pairs as an alternative to parentheses. Note that none of these constructions is a minus sign —, which has yet a different length and is produced only in math mode. (See Section A.4.)

- To generate opening and closing (double) quotation marks, use ‘‘ and ’’, i.e., two opening or closing (single) tics in a row, respectively. Do not use the symbol * for either of these punctuation marks.

- To specify a footnote, use the command $\footnote{Text of footnote.}$. The specified text will be placed at the bottom of the page and keyed to the text with an automatically generated number that starts with 1 at the beginning of the document.

- To start a new section, use the command $\section{Title of Section}$. In the article class, section numbers start with 1 at the beginning of the document and are automatically generated and incremented; the section number is included in the printed section title, which is displayed in a type size and font specified in the document class, and indentation will be suppressed in the first paragraph of each new section. [The commands $\subsection{}$ and $\subsubsection{}$ function in a similar way for subsections and subsubsections in the document. In the book class, the commands $\chapter{}$ and $\part{}$ (an aggregation of chapters) are also available.]

- To control hyphenation, insert the command $\hyphenat{point}$ at the point in a word where $\LaTeX$ is permitted to insert a hyphen as it fills and justifies lines. $\LaTeX$ has its own rules for hyphenation, but they are not infallible. Sometimes $\LaTeX$ needs help. Note that words containing even one indication of an optional hyphen will not be hyphenated at any other point(s) in the word.

- To suppress hyphenation of a single word altogether, place the word as the argument of an $\mbox{customize}$ command, e.g., $\mbox{customize}$.  

### A.4 Including Equations

Most scientific documents will have equations, all of which must be specified in $\LaTeX$’s math mode. Shifting from the default text mode to math mode can be accomplished in several ways. For short, unnumbered equations or mathematical symbols that appear in a line of text, the equation or symbol must be enclosed in one of the two constructions $(...)$ or $\$...\$, which are equivalent. If we want one or more displayed equations, with or without automatically generated equation numbers, we need to use the form $\begin{equation}\end{equation}$. Three standard environments use this construction. The displaymath environment centers a single equation on a line by itself, does not number the equation, and can be specified by the shortened form $\[\ldots\]$. The equation environment centers a single equation on a line by itself and numbers the equation automatically. The eqnarray environment—see The $\LaTeX$ Manual—is used for a string of equations and for long equations that will not fit on one line; its syntax allows vertical alignment of two or more displayed and numbered equations. To facilitate references to numbered equations produced in the equation and eqnarray environments, $\LaTeX$ provides the command $\label{ReferenceName}$ for defining a symbolic label within the environment and the command $\ref{ReferenceName}$ to permit referencing the equation by its number in the text. Further, the command $\pageref{ReferenceName}$ permits referencing the page on which the equation occurs in the text.²⁶ (Warning: The displaymath, equation, and eqnarray environments will generate error messages if they contain blank lines.)

²⁶When these references are used, the information written into the .aux file becomes important and the code must be processed by latex or pdflatex twice, once to write the correct information into that file and a second time to read and make use of that information. Other environments, such as table and figure also admit a $\label{}$ command and can make use of this capability for symbolic internal references to those components. The label command can also be used inside the argument of $\part$, $\chapter$, $\section$, $\subsection$, $\subsubsection$, and $\footnote{}$ and in the text following an $\item{}$ command in an enumerate environment to facilitate reference to these components of the document.
\LaTeX{} also has many symbols and structures, common in mathematical formulas, that can be used only in math mode. Subscripts and superscripts, roots, Greek letters, integral signs, summation signs, the \texttt{array} environment, and many more are available.\footnote{For a complete list of all symbols and the commands that create them, see Section 3.3 on Mathematical Formulas in \textit{The \LaTeX{} Manual}.} To reiterate, these symbols and structures can only be used if \LaTeX{} is in math mode. So, to insert the symbol $\Theta$ in the final output, we place the command \$\Theta\$ in the code.\footnote{In math mode, Greek letters can be produced by commands that simply name the letter, e.g., \texttt{\Theta} for $\Theta$ and \texttt{\omega} for $\omega$. Only the initial letter of the name is capitalized to produce an upper-case letter. Note, however, that commands for letters that are identical to Arabic letters, e.g., \texttt{\kappa}, do not exist.} (Here, the first dollar sign puts \LaTeX{} in undisplayed math mode and the second returns \LaTeX{} to text mode.) Since \LaTeX{} follows its own rules about spacing when in math mode, spaces in math mode are usually ignored. (A space, however, is necessary in such contexts as $\omega t$ to separate the command $\omega$ from the character t.) The commands $\,, \!$, $\,$, and $\;\,$ can be used in math mode to fine tune the spaces between elements in the equation by adding a thin space, a negative thin space, a medium space, and a thick space, respectively. It is, for example, customary to separate a differential element from what precedes and follows with a thin space, e.g., $(x+y)\,dx\,dy$ rather than $(x+y)\,dx\,dy$.\footnote{Technically and officially, the differential element $dx$ should be written with a \textit{Roman} d and an \textit{italic} x, i.e., $dx$, but this convention is rarely followed.}

Following the conventions for the type setting of mathematical text, all symbols in a math environment are automatically italicized. The most common departure from this convention occurs with the names of the standard mathematical functions, which are conventionally set in Roman type. Special math mode commands of the form $\texttt{\cos}$ and $\texttt{\sin}$ should be used for these functions.\footnote{See Table 3.9 in \textit{The \LaTeX{} Manual} for a complete listing.} Except for $\texttt{sc}$, the declarations shown in Table A.7 can also be used in math mode to specify text in something other than italic type but, in some contexts it may be preferable to use one of the commands $\texttt{\mathrm{\_\_\_\_}}$, $\texttt{\mathit{\_\_\_\_}}$, $\texttt{\mathbf{\_\_\_\_}}$, $\texttt{\mathtt{\_\_\_\_}}$, etc. described in Section 3.3.8 of \textit{The \LaTeX{} Manual}. Remember, however, that spaces are ignored in math mode, so a shift to Roman type with the command $\texttt{\mathrm{\_\_\_\_}}$, for example, to include a \textit{two}-word phrase, will result in the words running together unless the \textit{space} command $\texttt{\_\_\_\_}$ is used to insert an \textit{explicit} space between the words. Sometimes, the better means to insert Roman text in equations is to use the command $\texttt{\mbox{\_\_\_\_}}$ to escape temporarily to text mode.

Including bold characters in \textit{equations} is especially tricky. The command $\texttt{\mathbf{\_\_\_\_}}$ “emboldens” only \textit{some} characters in the processed formula and, in particular, leaves lower case Greek symbols in light face. The $\texttt{\boldmath{\_\_\_\_}}$ declaration causes \textit{everything} in math mode to be bold but cannot itself be \textit{invoked} in math mode. To include a bold face, lower case $\theta$, for example, we must use the construction \[ \ldots \texttt{\mbox{\_\_\_\_}} \texttt{\boldmath{$\theta$}} \ldots \] where the command $\texttt{\mbox{\_\_\_\_}}$, which \textit{is} valid in math mode, temporarily shifts to text mode so the $\texttt{\boldmath{\_\_\_\_}}$ declaration can be invoked and the math mode expression in the argument of the command $\texttt{\mbox{\_\_\_\_}}$ will then be made bold; the bold presentation is confined by the command $\texttt{\mbox{\_\_\_\_}}$ to the desired part of the total expression.

\section*{A.5 Including Lists}

Among the most commonly used environments are those that facilitate creation of lists, including the \texttt{itemize} environment for bulleted lists (such as the one in Section A.3.6) and the \texttt{enumerate} environment for lists in which the items are lettered or numbered in sequence (such as the one in Section A.17). Each environment is introduced and terminated with the standard construction $\texttt{\begin{\ldots}} \ldots \texttt{\end{\ldots}}$. Within the environment, each new item is introduced with the command $\texttt{\item{\_\_\_\_}}$. Thus, the structure might look like
\begin{itemize}
\item Text of first item.
\item Text of second item.
\item Text of third item.
\end{itemize}

The bullets and the numbers are, of course, generated automatically and, within the \texttt{enumerate} environment, the numbers are automatically incremented. Details will be found in Sections 2.2.4, 6.6, C.6.2, and C.6.3 in The \TeX\ Manual. Note in particular

- The optional argument for the command \texttt{\item}, which provides a means to override the automatic label for that item and replace it with an explicit stipulated label. For example, the command \texttt{\item[DMC]} will cause the item to be labeled ‘DMC’ and the command \texttt{\item[]} will suppress the label altogether. Note that, if the text of the item itself begins with something enclosed in square brackets, the command introducing that item must be written \texttt{\item{}} to prevent the text in square brackets from being interpreted as the desired label.

- The length parameters \texttt{\itemsep}, \texttt{\parskip}, and \texttt{\parsep}, which provide a means to override default spacings within the environment.

- The way to change the “bullets” in the \texttt{itemize} environment. The symbols used to label the “bulleted” items in the various levels of nested \texttt{itemize} environments are created by the commands \texttt{\labelitemi}, \texttt{\labelitemii}, \texttt{\labelitemiii}, and \texttt{\labelitemiv}. The default sequence for marking items at each level is •, – *, and ·. Each of these symbols can, however, be changed by redefining the corresponding command. For example, the command

\begin{verbatim}
\renewcommand{\labelitemii}{$\circ$}
\end{verbatim}

executed in the preamble will change the symbol – to ◦ for the second level in nested \texttt{itemize} environments.

- The way to change the form of the labels for each item in the \texttt{enumerate} environment. The labels at the various levels of nested \texttt{enumerate} environments are determined by invoking one of the commands \texttt{\theenumi}, \texttt{\theenumii}, \texttt{\theenumiii}, and \texttt{\theenumiv}, whose action is in turn determined from the value of the corresponding one of the counters \texttt{enumi}, \texttt{enumii}, \texttt{enumiii}, and \texttt{enumiv}. The default sequence for marking items at each level is ‘1.’, ‘(a)’, ‘i.’, ‘A.’, i.e., arabic number with period, lower-case letter in parentheses, lower-case Roman number with period, and upper-case letter with period. These defaults, however, can be changed with a command like

\begin{verbatim}
\renewcommand{\theenumi}{\Alph{enumi}}
\end{verbatim}

which will change the labels used at the first level in \texttt{enumerate} environments to upper-case arabic letters in parentheses. The command \texttt{\Alph} could be replaced with \texttt{\alph}, \texttt{\roman}, \texttt{\Roman}, or \texttt{\arabic} to translate the underlying counter into lower-case arabic letters, lower-case Roman numbers, upper-case Roman numbers, or arabic numbers, respectively. As it turns out, whatever is specified in the argument of the commands translating the counters, a period will be appended by \TeX\ at the first, third, and fourth levels, and enclosing parentheses will be supplied at the second level.

A.6 Including Tables

Creation of columnar arrangements is facilitated by \TeX\’s \texttt{tabular} environment, the details of which are involved and are fully described in The \TeX\ Manual. The \texttt{tabular} environment can
A.6. INCLUDING TABLES

Table A.9: Format for inserting a table.

\begin{table}
\caption{...}
\label{...}
\begin{center}
\begin{tabular}{clr}
Row 1, Col 1 & Row 1, Col 2 & Row 1, Col 3 \\
Row 2, Col 1 & Row 2, Col 2 & Row 2, Col 3 \\
Row 3, Col 1 & Row 3, Col 2 & Row 3, Col 3
\end{tabular}
\end{center}
\end{table}

be invoked anywhere in the code and the resulting table will be placed at the point at which the environment appears, perhaps even in the middle of a line of text. Conventionally, however, tables are placed at the top or bottom of the page,\footnote{Actually, the command \texttt{\begin{table}} has an optional argument (\texttt{\begin{table}[OptArg]}, which can assume any of the values h, b, or t for placement of the table at the place where the \texttt{table} environment appears (h, for here) or at the bottom (b) or top (t) of the page. In the absence of an explicit specification, \LaTeX\ makes its own decision about placement.} and \LaTeX\ provides the \texttt{table} environment to facilitate proper placement of a table,\footnote{If there is space on the current page when the \texttt{table} environment is encountered, the table will be placed on that page. Otherwise, the table will be placed on a subsequent page.} though the bare \texttt{table} environment contains nothing other than its name that suggests its use for tables. Any text at all can be incorporated in the \texttt{table} environment and will be placed on the page as would a table. Most commonly, however, the \texttt{table} environment will embrace a \texttt{tabular} environment defining the table itself and will use the command \texttt{\caption} to specify the caption of the table and the command \texttt{\label} to specify a symbolic label for use in referring to the table within the document.\footnote{The \texttt{\caption} and \texttt{\label} commands can be placed anywhere within the \texttt{table} environment. Captions will commonly be placed either above or below the captioned component. In some contexts (see the warning at the end of Section A.10), placing the caption above the item is preferable.} Frequently, the \texttt{tabular} environment will itself be embraced in a \texttt{center} environment to control the horizontal placement of the table on the page. Thus, the common structure for the code defining a table would be of the form shown in Table A.9.\footnote{\LaTeX\ automatically uses the specified caption as an entry for a list of tables. There is, however, a limit to the length of such entries. Especially long captions may generate error messages. To avoid this problem, we can—and sometimes must—exploit an optional argument to the command \texttt{\caption}\{\ldots\} which allows the user to dictate the entry made to the list of tables. Unless a list of tables is to be generated, some authors argue that we should routinely specify a null value for the optional argument by writing the command \texttt{\caption[\{\ldots\}}}, though we shall here not follow that recommendation. See \textit{The \LaTeX\ Manual} for details.} Frequently, the \texttt{tabular} environment will itself be embraced in a \texttt{center} environment to control the horizontal placement of the table on the page. Thus, the common structure for the code defining a table would be of the form shown in Table A.9.\footnote{See Sections 3.6 and C.10 in \textit{The \LaTeX\ Manual} for much more detail on ways to line things up in columns. Note particularly (1) the command \texttt{\multicolumn}, which provides for entries that span more than one column, and (2) ways to create horizontal and vertical rulings in the table.}

Here, individual elements in a row are separated from one another by ampersands, and the end of each row—except the last—is marked with the command \texttt{\}. In this example, the caption, including an automatically generated phrase and number of the form ‘Table 3:’, will appear above the table. Positioning the commands \texttt{\caption} and \texttt{\label} after the \texttt{center} environment would place the caption below the table. Whichever position is adopted, the command \texttt{\label} must appear after the command \texttt{\caption}. Wherever the command \texttt{\label} is positioned, it will place an entry in the \texttt{.aux} file so that the commands \texttt{\ref} and \texttt{\pageref} will function here as described for equations at the beginning of Section A.4.

The illustrative argument \texttt{clr} of the \texttt{tabular} environment requires a bit more explanation.
APPENDIX A. INTRODUCTION TO \LaTeX

In general, this argument will be a string of c’s, l’s and r’s, one for each column in the table. Each specifies the position (centered, left justified, right justified), respectively, of the entry in the corresponding column. In the example, there are three columns, with entries in the first column centered, entries in the second column left justified, and entries in the third column right justified. The argument is mandatory but will, of course, be a string appropriate to the table being constructed rather than the specific string clr.

By default, tables will be produced with no vertical or horizontal rulings. See Section C.10.2 in The \LaTeX Manual for a description of the means to add these rulings.

A.7 Including Illustrations

Graphics displays frequently appear in technical documents. Within \LaTeX, means to incorporate graphics include:

- A cut and paste method (Section A.7.1).
- A method for incorporating a properly constructed PostScript or PDF file that invokes the command \texttt{\includegraphics} from the package \texttt{graphicx} (Section A.7.2).
- A method that exploits the \texttt{tikz} package of macros that facilitate describing the graphical display and its formatting in the same way that \LaTeX itself facilitates describing the text and its formatting (Section A.7.3).
- The built-in \texttt{picture} environment (Section A.7.4), which has limited capability but is sometimes just the ticket for simple displays.

Most often, commands incorporating figures will be bracketed in a \texttt{figure} environment, which facilitates locating the illustration at the top or bottom of the current (or a following) page. In most cases, we need to know the final vertical extent of the figure in order to specify the size of an appropriate space. Further, the commands \texttt{\caption} and \texttt{\label} are available to caption the figure and to place appropriate entries in the .aux file so that the commands \texttt{\ref} and \texttt{\pageref} will function here as described in the first paragraph of Section A.4.

A.7.1 Using Cut and Paste

The cut and paste method requires only that \LaTeX be instructed to set aside appropriately sized and positioned space into which the illustration will be placed after the document has been printed. The segment of the code that creates and labels space for a figure will have the general form

\begin{verbatim}
\begin{figure}
  \caption{ ... }
  \label{ ... }
  \vspace{ ??.?truein }
\end{figure}
\end{verbatim}

As illustrated here, the caption, including an automatically generated phrase and number of the form ‘Figure 3:’, will appear above the space left for the figure. Positioning the commands \texttt{\caption} and \texttt{\label} after the command \texttt{\vspace} would place the caption below the figure. Note also that,

\footnote{We elect here to discuss only a very few of the numerous available packages, choosing those of the broadest applicability. Packages for drawing Feynman diagrams, organic molecules, musical scores, and many other displays are described in \textit{The \LaTeX Graphics Companion}.}

\footnote{The \texttt{figure} environment also admits the optional argument described in footnote 31.}
whichever position is adopted, the command `\label` must appear after the command `\caption`. For this method, you need only have printed copies of your figures. The format of the files used for the figures is irrelevant and the document can be produced with any of the methods described in Sections A.1.2 and A.1.3.

A.7.2 Using the `graphicx` Package

Incorporation of graphic images in a \LaTeX document is viewed by \LaTeX as the responsibility of the device driver. To enable that process, \LaTeX allows the passing of commands to the device driver. Both the format of such commands and the variety of capabilities thus available depends entirely on the device driver. In essence, this process uses the \TeX command `\special`, but the user is unaware of that underlying command because it is invoked behind the scenes.

The simplest means for importing graphical displays into a \LaTeX document—i.e., of commanding \LaTeX to pass the proper information on to the device driver—exploits the `graphicx` package, whose features are made available by including that package with the statement

\begin{verbatim}
\usepackage{graphicx}
\end{verbatim}

in the preamble of our \LaTeX code. After placing the command `\usepackage{graphicx}` in our preamble, we incorporate the graphical display itself by placing a statement like

\begin{verbatim}
\includegraphics[height=dimension]{filename}
\end{verbatim}

at the point where the PostScript or PDF figure defined in by the file `filename` is to be inserted. In general, the image would be centered horizontally on the page by inserting this command in a `center` environment, e.g.,

\begin{verbatim}
\begin{center}
\includegraphics[height=dimension]{filename}
\end{center}
\end{verbatim}

For example, the code

---

38 Appropriately translated, the substance of footnote 34 applies here to the entry placed automatically in the list of figures.

39 The ` graphics` package might also be used, but the syntax of the commands it defines is marginally less convenient than the syntax of the parallel commands in the `graphicx` package, i.e. the `extended graphics` package.

40 More generally, this statement admits an optional argument to the desired graphics driver explicitly. Frequently, that driver will be `dvips` and the statement would be `\usepackage[dvips]{graphicx}`, but numerous other drivers exist, and many may be installed at your site. See the `Local Guide` for more information. For ultimate flexibility in the formatting of graphics files, we elect to leave that choice to the default.

41 If you use `latex`, `dvips`, and `ps2pdf` as described in Section A.1.2 and item 2 in Section A.1.3 to produce PostScript and PDF documents, your figures must be available as `.ps` or `.eps` files, and—even if the file type is omitted in the `\includegraphics` command—the processing will correctly identify and incorporate the figures. If you seek to produce a PDF document directly with `pdflatex` as described in item 1 in Section A.1.3, then—even if the file type `.pdf` is omitted in the `\includegraphics` command—the processing will correctly identify and incorporate the figures. If your figures are PostScript, you must use the first route to the final document; if your figures are PDF, you must use the second route. Software to convert PostScript figures to PDF figures and vice versa is described in Section A.11. If both PostScript and PDF files exist for all figures and you omit the file type in the `\includegraphics` command, then you can use either `latex` or `pdflatex` for the processing.

42 Appropriately translated, the substance of footnote 34 applies here to the entry in the list of figures.
Figure A.1: Mesh surface representation of the irradiance produced by a square aperture.

\begin{figure}
\caption{Mesh surface representation of the irradiance produced by a square aperture.} \label{LATEX:irrad}
\begin{center}
\includegraphics[height=3.0truein]{diffract}
\end{center}
\end{figure}

will produce Fig. A.1 if the file \texttt{diffract} is stored in the current default directory.\footnote{Omission of the file type in the statement including the graph prepares the way to use either \texttt{latex-dvips} to produce a PostScript file using \texttt{latex} and \texttt{dvips} (Section A.1.2), in which case the file \texttt{diffract.ps} or \texttt{diffract.eps} will be used, or \texttt{pdflatex} to produce a PDF file (Section A.1.3), in which case the file \texttt{diffract.pdf} will be used. The appropriate file must, of course, be accessible in either case.} Here, the specification of the parameter \texttt{height} in the optional argument of the command \texttt{\includegraphics} causes the figure to be scaled to the specified dimension vertically and scaled by the same fraction horizontally to preserve its aspect ratio. As in the example in Section A.7.1, the figure here has been inserted before the command \texttt{\caption}, so that the caption will appear above the figure. Further, the command \texttt{\label}, which must follow the command \texttt{\caption}, defines a symbolic label that can be used within the document to refer to the figure.

\textbf{Warning for OCTAVE users:} All graphics toolkits in OCTAVE will produce proper on-screen graphs and will output PostScript files of these graphs that can be displayed with ghostview. If, however, the graph is to be incorporated in a $\LaTeX$ document by the procedures described above, graphs produced by all toolkits will be properly incorporated in the .\texttt{dvi} file but those produced by the \texttt{qt} toolkit \textit{may} not then translate with \texttt{dvips} to the subsequent PostScript file. When the latter is the objective, use \texttt{gnuplot} (or maybe \texttt{fltk}). A bit of testing may be necessary.

The command \texttt{\includegraphics} admits several optional arguments. The argument \texttt{width} can also be included. If \texttt{width} is specified \textit{instead of} \texttt{height}, the height will be scaled to preserve
the aspect ratio. If both \texttt{height} and \texttt{width} are specified, each will be respected and the aspect ratio of the figure may be distorted; specification of \texttt{neither} will cause the display to be produced in its original size. Arguments that allow overriding of the bounding box specified in the PostScript file and arguments that permit clipping a portion of a fuller figure are described in \textit{The $\LaTeX$ Graphics Companion}.

A.7.3 Using the \texttt{tikz} Package

The \texttt{tikz} package, which is routinely included in present-day $\LaTeX$ distributions, provides an assortment of $\LaTeX$ macros to facilitate the drawing of simple—and even complicated—figures. Basically, the statement \texttt{\usepackage{tikz}} in the preamble followed in the document itself by statements embedded in a \texttt{tikzpicture} environment will specify the desired graph and incorporate the graph in a PDF or Postscript file created as described in Sections A.1.2 and A.1.3.\footnote{Note that, when displayed on the screen with \texttt{xdvii} or \texttt{yap}—see Section A.1.4—the intermediate \texttt{.dvi} file produced in Section A.1.2 will \textit{not} contain the correct figures.} \texttt{Be aware that the code inserted in the \texttt{tikzpicture} environment describes the picture in the same way that $\LaTeX$ provides the text and its formatting. Just as one imagines the final formatted document as the $\LaTeX$ coding is constructed, so one has to imagine the resulting picture as one constructs the descriptive code to create it in the final document. \texttt{Tikz} \textit{is not} a WYSIWIG drawing program like \texttt{PAINT} or \texttt{TGIF}. One of its advantages over the route described in Section A.7.2, which imports figures produced in other programs, is that the route of this section guarantees that the fonts used in the figures will be identical to those used in the rest of the document.}

The graphics system provided by \texttt{tikz} is impressively elaborate and versatile. As such, learning its capabilities, especially its more sophisticated capabilities, will require substantial effort. Typing the command\footnote{The designation \texttt{pgf}—Portable Graphics Format—reflects the name of the engine underlying the entire system.} \texttt{texdoc pgf}

at a \textit{Shell} window to your operating system will probably bring up links to a number of manuals, including the main—and voluminous (1100-plus pages!)—manual in the file \texttt{pgfmanual.pdf}. Further, googling \texttt{tikz} will bring up links to numerous documents, including in particular a link to \texttt{pgfmanual.pdf} and a link to the much more compact tutorial-style introduction in the file \texttt{minimaltikz.pdf}. Once you have selected any of these items, you may in some operating systems have to look in your \texttt{Downloads} folder to access the item.

This section quickly orients you to the general features of \texttt{tikz} without in any way pretending to be complete. To that end, having included the statement\footnote{Some installations may also require inclusion of the packages \texttt{pgf} and \texttt{xcolor}.} \texttt{\usepackage{tikz}}

in the preamble of a document, we produce Fig. A.2 by placing the coding in Table A.10 at the appropriate point in the document itself. This coding briefly illustrates some of the more elementary capabilities of \texttt{tikz}. Note specifically the following:

\begin{itemize}
  \item Optional arguments to any \texttt{tikz} command are included in (square) brackets.
  \item By default, specified coordinates are expressed in centimeters, though any recognized $\LaTeX$ unit of length can be specifically stipulated. The optional argument \texttt{[x=1.0in,y=1.0in]} following \texttt{\begin{tikzpicture}} changes the default to inches. Once a figure has been defined, its size can be easily altered simply by changing this optional argument.
\end{itemize}
• Every statement must end with a semicolon. Any statement can be spread over several lines with only the last line so terminated.

• The statement `\draw [fill=black] (0,0) circle [radius=0.1];` draws a black-filled circle centered at the origin and having radius 0.1 inches, thus marking the origin of the coordinate system.\footnote{Note that the default unit—here inches—for radius can be overridden with an explicit stipulation, e.g., [radius=0.1cm].} Available colors include red, green, blue, cyan, magenta, yellow, and many others. You can also define your own color using a command like\footnote{The command `\definecolor` here used is defined within the `tikzpicture` environment and does not require use of the L\LaTeX\ package `color`.}

\begin{verbatim}
\definecolor{ColorName}{rgb}{r,g,b}
\end{verbatim}

where `ColorName` is the name you assign to the color and `r`, `g`, `b` specify the `r`, `g`, and `b` components that make up the color. Available shapes include rectangle, ellipse, and arc.\footnote{See the manuals for the details of how to specify the dimensions of these shapes.}

• The statement `\draw [<->, line width=2] (0,1.2)--(0,0)--(2.3,0);` draws a line connecting the specified points in the order given, i.e., draws the axes intersecting at the origin. Any number of points can be included in the path. The optional argument `->[->` places an arrow at both ends of the line,\footnote{The separate arguments `[->` and `<-` will place an arrow at the end or the beginning of the line, respectively.} and the argument `line width` species the weight of the line, by default, in points.\footnote{The argument `line width=2` could be replaced, among others, by `ultra thin`, `very thin`, `thin`, `thick`, `very thick`, and `ultra thick`, and the default unit could be overridden with an argument like `line width=0.1cm`.}

• Text is placed where specified with the statements `\node at (0,1.3) {\Large\bf y};` and `\node at (2.4,0.0) {\Large\bf x};`. Note that L\LaTeX\ stipulations of font size and style are recognized by tikz. Note also that the text is, by default, centered at the specified point. See the TIKZ manuals for ways to override that default positioning.

• The statement `\draw [ultra thick, domain=0:2.0] plot (\x, {sin(pi*\x r)} );` illustrates how to draw a smooth curve defined by a function, many of which are available within tikz. Note also the availability of the irrational number $\pi$ with the simple coding `pi`.\footnote{Another way to draw smooth curves is to use a calculational aid in another program to generate a probably long list of coordinates for many points on the curve and then to insert that list as the argument of a `\draw` command.}

• The construction

\begin{verbatim}
\foreach \x in {0.0,1.0,2.0} {
  \draw (\x,0.1)--(\x,-0.1);
  \node at (\x, -0.4){\x};;
}\end{verbatim}

creates a loop executing the statements enclosed in `\{\ldots\}` for each value in the list following the keyword `in`. The two loops in this code place tick marks on the axes and label those marks. Note the semicolons in this construction.

Clearly the command `\draw` is a versatile command that admits numerous embellishments through the use of keywords and/or optional arguments. Conveniently, tikz determines the size of the space to be used in the document to reflect the size of the picture as constructed with the measures provided in the tikz coding.

When features of the package tikz are invoked in the L\LaTeX\ code, producing the formatted document must be done carefully. If there are no complications (external figures, table of contents, internal references) to be incorporated, the simple statement
Figure A.2: An illustrative \texttt{tikz} figure. Note that specified colors will be converted to a gray scale unless the display device—screen or printer—can display colors.

![Illustrative tikz figure](image)

Table A.10: Coding to produce Fig. A.2.

```latex
\begin{figure}
\caption{An illustrative \texttt{tikz} figure.}
\label{LATEX:tigzfig}
\begin{center}
\begin{tikzpicture}[x=1.0in,y=1.0in]
\draw [fill=black] (0,0) circle [radius=0.1];
\draw [->, line width=2] (0,1.2)--(0,0)--(2.3,0);
\node at (0,1.3) {\Large\textbf{y}};
\node at (2.4,0.0) {\Large\textbf{x}};
\draw [ultra thick, domain=0:2.0] plot (\x, {sin(pi*\x r)});
\foreach \x in {0.0,1.0,2.0} {
  \draw (\x,0.1) -- (\x,-0.1);
  \node at (\x, -0.4) {\x};
}\foreach \y in {0.0,0.5,1.0} {
  \draw (-0.1,\y) -- (0.1,\y);
  \node at (-0.3, \y) {\y};
}\end{tikzpicture}
\end{center}
\end{figure}
```

\texttt{pdflatex filename}  
(\text{Default file type .tex})

\texttt{latex filename}  
(\text{Default file type .tex})

\texttt{dvips \(-o\) filename.ps \(-t\) letter filename}  
(\text{Default file type .dvi})

\texttt{ps2pdf filename.ps}  
will produce a PostScript and then a PDF file, though figures defined by \texttt{tikz} will not be properly
rendered in the intermediate .dvi file. If a table of contents or internal references or both are involved, two—and maybe three—passes through `latex` or `pdflatex` will be necessary. If there are PostScript figures and no PDF figures to be incorporated, only `latex` will work; if there are PDF figures (and no PostScript figures) or hyperlinks to be incorporated, only `pdflatex` will work. Finally, if there are both PostScript and PDF files defining figures, files of one type will have to be converted to the other type before either of these routes to a finished document will work.\footnote{See Section A.11 for details about that conversion.}

If you wish to short circuit learning detailed `tikz` code, you might wish to explore a WYSIWYG editor `TikzEdt` that provides a graphical interface for creating figures and translates that figure into the corresponding `tikz` code. This program, which is free, can be downloaded from the web site `www.tikzedt.org`. Documentation is also available at that site.

This section provides a woefully incomplete introduction intended more to wet your appetite than to make you an expert. The effort invested to study the manuals identified in the second paragraph of this section will be richly rewarded.

### A.7.4 Using the `picture` Environment and `pict2e` Package

Finally, we merely mention the `picture` environment, described in Sections 7.1 and C.14.1 of *The TeX Manual*, and the supplementary `pict2e` package, described in *The TeX Companion*. These components add several commands that facilitate the detailed construction of at least simple graphical displays.

### A.8 Including a Table of Contents, a List of Figures, and a List of Tables

A table of contents, a list of figures, and a list of tables can be included in the formatted document by the commands `\tableofcontents`, `\listoffigures`, and `\listoftables`. Each

- results in the output of an auxiliary ASCII file of information with file type `.toc`, `.lof`, and `.lot`, respectively,
- requires a second pass through `latex` or `pdflatex` to incorporate the list in the finished document,
- places the list in the output at the point at which the command appears.

When these commands are invoked, each of the components of the text defined by the commands `\part`, `\chapter`, `\section`, `\subsection`, and `\subsubsection` will automatically generate a line in the `.toc` file.\footnote{Note that not all of these options are available in all document classes. For example, `\part` and `\chapter` are available only in the `book` class. Note also the counters `secnumdepth` and `tocdepth` that control the depth to which sections are numbered and the depth to which sections are catalogued in the table of contents. The default values of these counters can be overridden using `\setcounter` in the preamble.} Each use of the `\caption` command in a `figure` or `table` environment will automatically generate a line in the `.lof` or `.lot` file. Then, in the second pass—which will always be necessary—through `latex` or `pdflatex`, these files are read and each line results in an entry in the corresponding list.

Two commands allow explicit entry of information into one or another of these files at the point where the command is inserted in the TeX source file, specifically

- the command `\addcontentsline{file}{unit}{entry}`
- the command `\addtocontents{file}{text}`
Here, file is one of toc, lof, and lot, unit is one of part, chapter, section, subsection, or subsubsection if file is toc, figure if file is lof, and table if file is lot.

These two commands have quite different effects. The first adds a bona fide entry to the corresponding file. For example, the line

\addcontentsline{toc}{subsection}{Specially marked point}

will add to the table of contents an entry that is left-justified at the subsection level, contains the text “Specially marked point”, and includes a row of dots and the appropriate page number. The second merely adds text—no dots or page number—to the corresponding file. For example, the line

\addtocontents{toc}{\vspace{24pt}This is a note.\vspace{24pt}}

inserts a note preceded and followed by a bit of extra vertical space. Any textual note is left-justified as dictated by the section level at which it appears in the document.

Additional details about the issues discussed in this section are laid out in Sections 4.1 and C.4 of The \LaTeX Manual. Note, in particular, the optional argument in the command \caption and the several sectioning commands (chapter, section, …), which provide control over the entry each command places in the table of contents, list of figures, and list of tables.

A.9 Including an Index

Among many capabilities, \LaTeX is able to generate an index, though not automatically. In essence, we must place the commands

\usepackage{imakeidx}
\makeindex

in the preamble and the command

\printindex

at the point in the code at which the index is to be printed—usually the very end. Then, we indicate what items are to be indexed by placing commands like

\index{Item to be indexed}

at appropriate points throughout the document. (The detailed structure of the argument to the command \index is described in Section 4.5 and Appendix A of The \LaTeX Manual.) When the file filename.tex is processed by latex or pdflatex, the presence of these commands results in the addition of three auxiliary ASCII files named

- filename.idx containing one line for each index entry,
- filename.ilg containing a log of messages created when, behind the scenes, the idx file is automatically processed through makeindex, and

\footnote{If you are creating a linked document, this entry in the table of contents will also appear in the navigation panel of that document.}

\footnote{Earlier versions of \LaTeX used the package makeidx, which is still available. We here recommend using imakeidx because it makes available a few more features than are available in the previous package and it eliminates the need to run an auxiliary program to format the index.}
APPENDIX A. INTRODUCTION TO \LaTeX

- `filename.ind` produced by `makeindex` and containing the formatted index that is then read by `\printindex` to create the actual index in the document.

With `imakeidx`, the index is automatically created and incorporated in the document because an execution of `makeindex` is inserted automatically at the appropriate point when `latex` or `pdflatex` is run. Errors and warnings in that step are compiled in the `.ilg` file (which should be examined because on-screen display of those glitches may well scroll by to quickly to be comprehended).\textsuperscript{58}

The above-described process yields an index in the default format. Numerous options can be exploited by adding optional arguments in the `\makeindex` command. For example, the command

\begin{verbatim}
\makeindex [options=-s ind_style, intoc]
\end{verbatim}

stipulates that the style defined by the file `ind_style.ist` should be invoked and the index should be provided—argument `intoc`—with an entry in the table of contents. The construction of the style file is fully described in Section 12.4 in *The \LaTeX Companion* as identified in Section A.18.

To assist in constructing the index, an auxiliary package called `showidx`, which works both with `latex` and with `pdflatex`, can be invoked to print the index entries on each page as marginal notes. This package is invoked simply by including the command

\begin{verbatim}
\usepackage{showidx}
\end{verbatim}

in the preamble, though it appears as if this command must be inserted after the `\makeindex` command described above. Unfortunately, when printing in a 6”×9” area on 8.5”×11” paper (`letterpaper`), the marginal insertions of index entries will extend outside the edges of the page and, in `.ps` and `.pdf` files, the portions that are off the page will not be displayed in `ghostview` or `acroread`. Conveniently, if the `.dvi` file is displayed on the screen with `xdvi` (UNIX) or `yap` (Windows), those marginal notes will be visible in their entirety, even if they extend beyond the boundaries of the page, though that route is available only if figures are provided in `.ps` or `.eps` format. For the purposes of checking the index entries, one workaround is to shrink the area of the page in which printing of the text is allowed. For example, for this document, replacing the commands

\begin{verbatim}
\setlength{\textwidth}{6.0truein}
\setlength{\oddsidemargin}{0.5truein}
\setlength{\evensidemargin}{0.0truein}
\end{verbatim}

in the preamble with the commands

\begin{verbatim}
\setlength{\textwidth}{4.0truein}
\setlength{\oddsidemargin}{1.0truein}
\setlength{\evensidemargin}{1.0truein}
\end{verbatim}

will shrink the text width and reset the margins so there will be space for the marginal notes. To be sure, the pagination of the text will also change, but at least the marginal notes will be visible, even in the `.ps` and `.pdf` files. In different situations, tampering with the above illustrated length parameters and, perhaps also, tampering with the length parameters `\marginparwidth` (which sets the width of the marginal boxes) and `\marginparsep` (which sets the space between the text and the boxes) as well as seeking ways to shrink the entire output including marginal notes on each page as a unit may yield fruit.\textsuperscript{59}

\textsuperscript{58}With `makeidx`, a separate explicit processing of the `.idx` file with `makeindex` to produce the `.ind` file and a further pass through `latex` or `pdflatex` was necessary to include the index in the document.

\textsuperscript{59}For example, the `-x` option to `dvips` can scale the size of each page output to the `.ps` file.
A.10 Including Hyperlinks

Properly constructed, a \LaTeX source file can be used to create a PDF document that includes a navigation panel and hyperlinked references for easy viewing with a compatible PDF viewer. Only a few changes need be made to the source files so far described to turn entries in the table of contents, internal references in the body of a document, and page references in an index into hyperlinks to the appropriate points in the document, though these features will appear only in PDF files created via the route described in item 1 in Section A.1.3 above. Specifically,

- the \texttt{\documentclass} statement at the beginning of the file must not specify any particular driver, i.e. should be of the form \texttt{\documentclass\{Class\}}.
- if used at all, the \texttt{\usepackage\{graphicx\}} command in the preamble must not include any options specifying a graphics driver.
- figures must be provided as PDF files or defined using the package \texttt{tikz}. PostScript descriptions of figures will not be properly rendered.
- the command \texttt{\usepackage\{hyperref\}}, perhaps with some optional arguments, must be added in the preamble and, to make sure that other included packages do not overwrite redefinitions made by \texttt{hyperref}, should be the last package included.

This edited source file is then processed by \texttt{pdflatex} (Section A.1.3) at least twice to include hyperlinks, and tables of contents, figures, and tables. With these simple additions, a navigation panel in the PDF file containing chapter, section, subsection, and ... subdivisions of the document will be created. Further, all entries in the tables of contents, list of figures, and list of tables, all internal references created with \texttt{\ref} commands, and all index entries will be converted into hot links within the document.

The command \texttt{\usepackage\{hyperref\}} in the preamble is the minimum necessary to create a hyperlinked PDF file. By default, hot links in the document will be displayed in a red box. To override that default, one can exploit options in the \texttt{\usepackage} command. For example, the statements,

\begin{verbatim}
\usepackage{color}
\definecolor{MyPurple}{rgb}{0.577,0.000,1.000}
\usepackage[colorlinks=true,linkcolor={MyPurple},bookmarksnumbered=true,linktocpage=true]{hyperref}
\end{verbatim}

in the preamble of the \LaTeX source file will set the stage for automatic creation of hyperlinks for all references, setting the color of those hyperlinks to \texttt{MyPurple} as specified by the RGB values in the \texttt{\definecolor} statement and removing the enclosing box, stipulating that the bookmarks in the navigation panels should include chapter and section numbers, and stipulating that page numbers rather than section titles should be the linked entries in the Table of Contents.\textsuperscript{60} The final PDF file is then produced by processing the source file with \texttt{pdflatex} (Section A.1.3)—at least twice and perhaps three times. To avoid error messages, all auxiliary files hanging over from a previous processing of the source file \textit{without} hyperlinks through \texttt{latex} or \texttt{pdflatex} should be deleted before processing the file with hyperlinks.\textsuperscript{61}

The files created in the user’s directory when a properly constructed \texttt{.tex} file with hyperlinks, index, and table of contents is processed through \texttt{pdflatex} to produce a linked PDF file are\textsuperscript{62}

\begin{itemize}
\item The default value of the \texttt{bookmarksnumbered} and \texttt{linktocpage} options is \texttt{false}.
\item See item 25 in Section A.17 for ways to achieve this deletion.
\item We here assume you are making an index and using \texttt{imakeidx}. The sequence and auxiliary files will be different if you are not making an index and/or if you are using \texttt{makeidx}.
\end{itemize}
• (after first pass through pdflatex) .aux, .idx, .ilg, .ind, .log, .out, .pdf, and .toc. The .idx, .ilg, and .ind files will be present only if you are creating an index with makeidx. The .out file contains information about hyperlinks; the rest are as described in item 1 of Section A.1.3, in Section A.8, and in Section A.9. At this point, the PDF file does not have the navigation panel or the table of contents but it does have the index. All of these files are ASCII text files, though the PDF file may contain some non-printing characters. As such they can all be displayed in an available text editor and understood.

• (after the second pass and, if necessary, the third pass through pdflatex) the same as in the previous bullet, though many of those files have been updated. At this point, the navigation panel, the internal hyperlinks, and the index have all been created.

A WARNING: Links to figures and tables will point to the line containing the caption for the figure or table. Captions for these items are perhaps better placed above the item rather than below the item.

A CONVENIENCE: If the commands \documentclass and \usepackage{graphicx} are phrased without options and the files in all \includegraphics commands are presented without file type, then the file processed with latex will look for .ps and .eps graphics files and the file processed with pdflatex will look for .pdf files. All files to be sought must, of course, exist. This feature makes it easy to process the same source file both with latex and with pdflatex.

A.11 Converting .eps and .ps Files to .pdf

When files describing figures are created, it is often easier to output those files from the creating program as .eps or .ps files rather than as .pdf files. Unfortunately, if you desire to produce the final output with pdflatex, files describing pictures need to be .pdf. PostScript files must therefore be converted into PDF before running pdflatex. Short of asking someone who already knows how to achieve that transfer, rummaging on the web for guidance may be a frustrating experience. Once you find the right tools, however, the process is quite simple. Basically, to convert the files figure.eps and figure.ps to a useful PDF file involves the two steps

1. ps2pdf figure.eps
   ps2pdf figure.ps
   (which will yield figure.eps.pdf)  
   (which will yield figure.pdf)

2. pdfcrop figure.eps.pdf
   pdfcrop figure.pdf
   (which will yield figure.eps-crop.pdf) or
   (which will yield figure-crop.pdf)

Here, step 1 creates the .pdf file and step 2 removes extraneous white space around the perimeter of the first-created .pdf file. Second arguments to both ps2pdf and pdfcrop can be used to relieve the awkwardness of the file names. To simplify the process, the MS DOS batch files listed in Section A.A.1, respectively, and the UNIX Shell scripts listed in Section A.A.2, respectively, exploit the second argument for both ps2pdf and pdfcrop to control the file names and, when executed with statements like

<table>
<thead>
<tr>
<th>MS DOS</th>
<th>UNIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>ceps2pdf figure</td>
<td>./ceps2pdf figure           (for .eps file)</td>
</tr>
<tr>
<td>cps2pdf figure</td>
<td>./cps2pdf figure            (for .ps file)</td>
</tr>
</tbody>
</table>

63 See Section A.11 for a means to convert .ps and .eps files to .pdf and vice versa.
will (1) leave in the involved directory a cropped PDF file whose name `figure.pdf` differs from that of the original PostScript in only the file type and (2) remove any temporary files created along the way.

The above procedure is a bit tedious if you have more than a few files to convert. If you work in a *Command* window (Windows) or a *Shell* window (UNIX), the steps

- Create a temporary directory and create the several files listed in Section A.A.1 (Windows) or A.A.2 (UNIX) and the file listed in Section A.A.3 (Windows and UNIX) into that directory. These files can be created by direct typing or they can be copied from the directory `$HEAD/tex`.

- Copy all `.ps` or all `.eps` files to be converted into that temporary directory. *This step guards against disaster should the process to come be run in the initial directory and fail.*

- Create a file containing a list of the names of the `.eps` or `.ps` files in the directory. The statements

  ```
  MS DOS
  dir /b *.eps > dir.txt
  dir /b *.ps > dir.txt
  ```

  ```
  UNIX
  ls -l *.eps > dir.txt
  ls -l *.ps > dir.txt
  ```

  utilizing the option `/b` in Windows and the option `-l` (one, not el) in UNIX produce the desired file (filename, including file type).

- Strip the file type by running the python program `ExtractFileName.py` (Section A.A.3). Here the same code works in both Windows and UNIX. This action will create the file `nameonly.txt` containing only the names of the files to be converted.

- Effect the conversions by running the script `rdfile` with statements like

  ```
  MS DOS
  rdfileeps (for eps files)
  rdfileps (for ps files)
  ```

  ```
  UNIX
  ./rdfileeps (for eps files)
  ./rdfileps (for ps files)
  ```

  This action will convert each file in turn from `.eps` or `.ps` to `.pdf` and leave no intermediate files in the directory.

- Delete the `*.eps` or `*.ps` files in the temporary directory.

- Move the `*.pdf` files to the directory from which the `.eps` and `.ps` files were moved temporarily, leaving both the original `.ps` and `.eps` files intact but adding the `.pdf` files, so either `latex` or `pdflatex` can then be used to create the final document.

will accomplish that conversion.

The reverse process of converting a `.pdf` file to `.ps` or `.eps` format is easier. Specifically, the statements

```
pdftops FileName.pdf FileName.ps
pdftops -eps FileName.pdf FileName.eps
```

will effect this conversion. The task of creating scripts to automate this process when many files are to be converted is left to the reader.
A.12 Using Conditional Expressions in \LaTeX

The \LaTeX package \texttt{ifthen} provides a capacity to include text and \LaTeX commands conditionally, i.e., depending on the state of a Boolean flag. If the package is to be used, the preamble of the \LaTeX source file must contain the command

\usepackage{ifthen}

Once that command has been processed, we must—also in the preamble—then define one or more Boolean flags and set their values with commands like

\newboolean{FlagName} \setboolean{FlagName}{FlagValue}

where \texttt{FlagName} can be any name that does not conflict with names already used and \texttt{FlagValue} will be either \texttt{true} or \texttt{false}. Finally, at the point in the source file where some text is to be included conditionally, we would place the command

\ifthenelse{\boolean{FlagName}}{Text to be inserted if \texttt{FlagValue} is \texttt{true}.}{Text to be inserted if \texttt{FlagValue} is \texttt{false}.}

Either text can be null, in which case the opening and closing braces should still appear side by side ({}), and either text can include \LaTeX commands—which will be executed—and can spread over several lines. In particular, the text can include commands to input additional files. Indeed, the customization in \textit{CPSUP} is achieved by including or excluding files depending on the state of several flags. For example, in the preamble of the \LaTeX source file for \textit{CPSUP}, the command

\usepackage{ifthen}

appears and several flags are defined and set with commands like, for example,

\newboolean{LATEX} \setboolean{LATEX}{true}

Finally, the command

\ifthenelse{\boolean{LATEX}}{\input{FileName} }{ }

is placed at the point where the file containing the \LaTeX portions of this book would be inserted. If the file identified is in the working directory, only its name need be included; otherwise, a full path—absolute or relative—must be specified. At this point, the \LaTeX Appendix will be included if the flag is \texttt{true} and omitted if the flag is \texttt{false}.

More complicated logical operations can be constructed with the operators \texttt{\and}, \texttt{\or}, and \texttt{\not}. For example, if the Boolean flags \texttt{NUMREC} and \texttt{FORTRAN} are defined and set, then the composite statement

\ifthenelse{\boolean{NUMREC} \and \boolean{FORTRAN} }{\input{FileName} }{ }

is constructed.
would include filename only if both NUMREC and FORTRAN are true.

In the effort to render files processable with either latex or pdflatex and also to create easily either a printable document or a linked document, the package hyperref must be invoked only when a linked document is desired. In order to avoid manual editing of the source file, one can insert in the preamble the statements

\usepackage{ifthen}
\newboolean{PRINT}
\typeout{} \typein\[trueorfalse\]{true (for print), false (for linked):}
\ifthenelse{ \equal{trueorfalse}{true} \or \equal{trueorfalse}{false}}{ \setboolean{PRINT}{trueorfalse} }{\typeout{} \typeout{Must be either "true" or "false". Try again.} \end{document}}

which define the boolean variable PRINT, ask for entry of either true or false at execution time, and terminate execution if an invalid value is entered. Then, the lines

\usepackage{color}
\ifthenelse{\boolean{PRINT}}{}{\definecolor{MyPurple}{rgb}{0.577,0.000,1.000} \usepackage[colorlinks=true, linkcolor={MyPurple}]{hyperref}}

also in the preamble, will include the package hyperref, but only if PRINT is false.

One further conditional command included in latex and pdflatex without adding a package facilitates testing whether a file exists. For example, the statement

\IfFileExists{test.tex}{\input{test.tex}{}}

inputs test.tex if it exists and simply moves on to the next statement if it doesn’t exist. This statement is useful if the file to be input is created in the first pass through latex or pdflatex and then read in a subsequent pass. The “false” clause could alternatively be used to print an error message and terminate execution, e.g.,

\IfFileExists{test.tex}{\input{test.tex}{\typeout{File not found} \end{document}}}

A.13 Error Messages Generated by \LaTeX

However carefully the code is created, sooner or later \LaTeX will generate an error message, which will have the general form

! error identification
1. number text read
   text not read
?

The exclamation point is followed by a message that explains the nature of the error. The lower case l (el) is followed (1) by the number of the line in the code at which the problem is detected

\textsuperscript{64}Inclusion of the package ifthen can be omitted if it had been entered previously for other reasons.
and (2) the text surrounding the problem. Finally, the question mark prompts for user input. A simple \texttt{RETURN} at the question mark will instruct \LaTeX{} to continue to read the file unless a fatal error has occurred (though \LaTeX{} may be forced to make assumptions in order to continue); entry of the character \texttt{x} (followed by \texttt{RETURN}) will exit immediately from the program. Many errors are caused by a missing delimiter or by failure to specify a switch to math mode \texttt{before} a mathematical expression or back to text mode \texttt{after} a mathematical expression. All of the reported errors must be identified and fixed \texttt{(by editing the code)} before \LaTeX{} will produce the desired finished product. A full description of the messages that may be produced as well as an enumeration of other user inputs at the ? will be found in Chapter 8 in \textit{The \LaTeX{} Manual}.

### A.14 The Page Previewer for \texttt{.dvi} Files

Since \LaTeX{} is not wysiwyg, we must use a page previewer to see exactly what our document looks like without printing it out many, many times. The \textit{Local Guide} describes the page previewer available at your site. UNIX systems usually provide a version of \texttt{xdvi} and Windows systems usually provide a version of \texttt{yap}, but many such previewers exist and, on personal computers especially, the previewer may be accessed by a mouse click in a GUI that also gives access to other components of the \LaTeX{} package. To use this previewer, we first create the \texttt{.dvi} file by running our code through \LaTeX{} as described in Section A.1. Then we enter a command like \texttt{xdvi filename} or \texttt{yap filename} (The default extension for the input file is \texttt{.dvi}.) or select an appropriate item from a menu \texttt{(as described in \textit{The Local Guide})}. Presently, the first page of the document will appear on the screen. Some versions of \texttt{xdvi} display an array of buttons along the right edge of the \texttt{Xdvi} window; other, probably newer, versions replace those buttons with drop-down menus and icons in a toolbar. Clicking \texttt{ML} on any of the various buttons or selecting items from one of the menus affects the associated action. For \texttt{xdvi} and its clones, possible actions include the options shown in Table A.11. In addition, (1) moving the cursor to any point in the window and pressing (and holding) any of the three mouse buttons will generate a magnified version of a region whose size is determined by which button is pushed and (2) some versions of \texttt{xdvi} have a window along the left edge in which you can click \texttt{ML} on a page number to request immediate display of the selected page—though note that these numbers simply count pages from the beginning of the displayed document and may not correspond to the numbers actually printed on the pages in the document.

Beyond the controls made available through buttons and/or menus around the periphery of the \texttt{Xdvi} window, \texttt{xdvi} responds to a number of keystrokes issued when the the cursor is moved into the area displaying text. Some of these are identified in Table A.12; all are described in the \texttt{xdvi} documentation, e.g., the UNIX \texttt{man} page.

In some environments, a GUI may provide an easy way to use the mouse to work on \LaTeX{} code, process it repeatedly with \LaTeX{}, and examine the changes with an on-screen previewer each time. In the absence of a GUI giving access to all of these features (text editor, \LaTeX{}, screen previewer, printer, ...), one convenient strategy for using a text editor \texttt{and} a page previewer simultaneously involves invoking the previewer with a command like

\texttt{xdvi filename} (UNIX) \texttt{or yap filename} (Windows)

\footnote{The point at which \LaTeX{} encounters difficulty is conveyed by the downward displacement of the text after that point, though that point is not always where the offense actually occurs.}

\footnote{Yet Another Previewer}

\footnote{Additional information about \texttt{xdvi} on UNIX systems can be found in the on-line help, accessed in many systems by typing the command \texttt{man xdvi}.}

\footnote{If explicit specification of the input file is omitted, some implementations of \texttt{xdvi} will bring up a browser in which the desired \texttt{.dvi} file can be selected. Other versions may open the most recently displayed file, in which case the browser can be invoked by selecting ‘Open’ from the \texttt{File} menu.}
Table A.11: Some of the features available through selection from menus (first column) in some versions of \textsc{xdvi}, through clicking ML on buttons in other versions of \textsc{xdvi}. The features actually available by these means may vary from version to version of \textsc{xdvi}.

<table>
<thead>
<tr>
<th>Menu</th>
<th>Button</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILE→Reload</td>
<td>Reread</td>
<td>Rereads current .dvi file.</td>
</tr>
<tr>
<td>FILE→Quit</td>
<td>Quit</td>
<td>Quits previewer.</td>
</tr>
<tr>
<td>ZOOM→Shrink by 1</td>
<td>100%</td>
<td>Displays text at full size (quite large).</td>
</tr>
<tr>
<td>ZOOM→Shrink by 3</td>
<td>33%</td>
<td>Displays text at 33% of full size.</td>
</tr>
<tr>
<td>ZOOM→Shrink by 4</td>
<td>25%</td>
<td>Displays text at 25% of full size.</td>
</tr>
<tr>
<td>ZOOM→Shrink by 6</td>
<td>17%</td>
<td>Displays text at 17% of full size.</td>
</tr>
<tr>
<td>NAVIGATE→First Page</td>
<td>First</td>
<td>Moves to first page.</td>
</tr>
<tr>
<td>NAVIGATE→Page-10</td>
<td>Page-10</td>
<td>Moves to the tenth page before the one displayed.</td>
</tr>
<tr>
<td>NAVIGATE→Page-5</td>
<td>Page-5</td>
<td>Moves to the fifth page before the one displayed.</td>
</tr>
<tr>
<td>NAVIGATE→Prev</td>
<td>Prev</td>
<td>Moves to previous page</td>
</tr>
<tr>
<td>NAVIGATE→Next</td>
<td>Next</td>
<td>Moves to next page.</td>
</tr>
<tr>
<td>NAVIGATE→Page+5</td>
<td>Page+5</td>
<td>Moves to the fifth page after the one displayed.</td>
</tr>
<tr>
<td>NAVIGATE→Page+10</td>
<td>Page+10</td>
<td>Moves to the tenth page after the one displayed.</td>
</tr>
<tr>
<td>NAVIGATE→Last Page</td>
<td>Last</td>
<td>Moves to last page.</td>
</tr>
<tr>
<td>OPTIONS→PostScript</td>
<td>View Page</td>
<td>Allows selection among two or more options, including displaying figures (the initial state) and replacing them with properly sized rectangles.</td>
</tr>
<tr>
<td>FILE→Open</td>
<td>File</td>
<td>Brings up a browser for selection of new .dvi file.</td>
</tr>
</tbody>
</table>

Table A.12: Keystrokes for controlling \textsc{xdvi}.

<table>
<thead>
<tr>
<th>Key</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>Size text to fit window.</td>
</tr>
<tr>
<td>1s</td>
<td>(one-s) Largest text.</td>
</tr>
<tr>
<td>ns</td>
<td>Apply shrink factor $n$.</td>
</tr>
<tr>
<td>g</td>
<td>Move to last page.</td>
</tr>
<tr>
<td>1g</td>
<td>(one-g) Move to first page.</td>
</tr>
<tr>
<td>ng</td>
<td>Move to (absolute) page $n$.</td>
</tr>
<tr>
<td>n</td>
<td>Move to next page.</td>
</tr>
<tr>
<td>p</td>
<td>Move to previous page.</td>
</tr>
<tr>
<td>pp</td>
<td>Advance $n$ pages.</td>
</tr>
<tr>
<td>np</td>
<td>Back up $n$ pages.</td>
</tr>
<tr>
<td>q</td>
<td>Exit program</td>
</tr>
</tbody>
</table>

which will start \textsc{xdvi} or \textsc{yap} but detach it from the launching \textit{Shell} window, leaving the \textit{Shell} window free for other uses. Then, with the \texttt{.tex} file in a text editor detached from the launching \textit{Shell} window, we can edit the text, save the edited version, use the \textit{Shell} window to invoke \textsc{\LaTeX} on the edited file, and then simply click ML at an appropriate point in the displayed text to instruct \textsc{xdvi} or \textsc{yap} to reread the .dvi file.\footnote{In some environments, rereading an updated file will happen automatically without an explicit user request.} Thus, the effect of each new set of edits can be examined quickly without repeatedly starting and exiting from the previewer.

A.15 The Spell Checker in UNIX\footnote{To the author’s knowledge, no comparable stand-alone spell checker exists for Windows.}

The programs \texttt{ispell} and \texttt{aspell}, either or both of which may be part of your \textsc{\LaTeX} distribution, are common spell checkers. If either is installed at your site (see the \textit{Local Guide}), it can be invoked
with a command like\footnote{Additional information about \texttt{ispell} or \texttt{aspell} can be found by typing the command \texttt{ispell} with no arguments or the command \texttt{aspell help}. Even more detailed information may be available in the on-line help, accessed in many systems by typing the command \texttt{man ispell} or the command \texttt{man aspell}. Googling \texttt{ispell} or \texttt{aspell} will surely also provide links to detailed descriptions.}

\texttt{ispell filename} \hspace{1em} or \hspace{1em} \texttt{aspell check filename}

or perhaps with a mouse click in a GUI. For files with extension .\texttt{tex}, \texttt{ispell} enters its \TeX{} mode and will not identify every \TeX{} or \L{a}\TeX{} command as a misspelled word. Note, however, that these commands are simply \emph{ignored} by \texttt{ispell} and \texttt{aspell}; their correctness or legitimacy as commands to \L{a}\TeX{} is \emph{not} assessed.

\section{A.16 A Sample Document}

The following sample document demonstrates some of the commands explained above and also introduces some useful commands that have not yet been mentioned. (The explanatory \emph{comments} following the \texttt{\%} sign can be included in the code but have no effect on the output produced.)

\begin{verbatim}
\documentclass{article} % Mandatory command; select
% default 10 point type
\setlength{\textwidth}{6truein} % Set width of text area
\setlength{\oddsidemargin}{0.25truein} % Center text on 8.5'' width
\setlength{\parindent}{40pt} % Set paragraph indent
\setlength{\parskip}{6pt plus 1pt minus 1pt} % Specify extra space
% between paragraphs, allowing \LaTeX{} to adjust
% space 1 point up or down
\begin{document} % Mandatory command
\begin{center} % Large, bold, centered title
{\Large\bf A Sample Document for Your Perusal}
\end{center}
\begin{flushleft} % Two lines; flush left
{\em Author/}: J.~Q.~Student \ \ % \ \ forces a new line
% Note italic correction
{\em Date/}: \today
\end{flushleft}

In this document we have used the \texttt{\setlength} commands to modify
\LaTeX{}'s default page setup to make fuller use of an 8.5'\times11'' page.\footnote{Note the special command $\backslash$tt \LaTeX{}}
provided for the display of the \LaTeX{} `logo'.

The title could have been generated using the three commands
\texttt{\title{...}+}, \texttt{\author{...}+}, and
\texttt{\date{...}+} before the \texttt{\begin{document}+}
\end{verbatim}
command---i.e., in the preamble---followed by the command \verb*+\maketitle+ after the beginning of the document. The command \verb*+\begin{verbatim} +? will cause LaTeX to print whatever is between the plus signs (which could be virtually any specified character) exactly as it is. The asterisk is optional, and it makes LaTeX highlight the spaces that occur inside the \verb+verbatim+ environment.

\noindent Notice that a blank line creates a new paragraph. The \verb+\noindent+ command \emph{suppresses} the standard paragraph indentation. In this special space we will also demonstrate some math environment operations. Consider the equation
\begin{equation}
\int_{0}^\infty e^{-x}, \, \text{dx} = 1, \\
\label{LATEX:demo}
\end{equation}
where a medium sized space, specified by the command \verb+\+:, is put between the integral sign $\int_{0}^{\infty}$ and the integrand and a small space, specified by the command \verb+\,,+ is inserted between the integrand and the $\text{dx}$ in Eq.~\eqref{LATEX:demo}.

The equation
\begin{equation}
\frac{\partial e^{x_0 y^2} }{ \partial x_0 } = y^2 e^{x_0 y^2}
\end{equation}
demonstrates partial derivatives and fractions. Note the automatic generation and placement of equation numbers in the output. Also note that pages are automatically numbered, though the actual printing of page numbers can be suppressed with the commands \verb+\pagestyle+ and \verb+\thispagestyle+.

\end{document} % Mandatory command

The output produced when this short sample file is processed through \LaTeX{} and \dvips{} and then printed is shown in Table A.13, though the page header on that page is not included in the output. Note that, to obtain the proper internal reference to the first equation, this document must be processed \textit{twice} by \LaTeX{}. The file itself is named \texttt{texsample2.tex} and can be copied from the directory $\$HEAD/tex$.

\section*{A.17 Miscellaneous Other Features}

\LaTeX{} has an enormous number of additional features beyond those enumerated above. In particular, be aware of

1. All of the environments listed in Section A.3.4.

2. All of the packages listed in Section A.3.5.

3. The command \verb+\today+, which returns today’s date in the form “month date, year”.

4. The command \verb+\newcommand+, which permits us to define commands supplementing the standard commands. For example, the commands
\begin{verbatim}
\newcommand{\beq}{\begin{equation}}
\newcommand{\eeq}{\end{equation}}
\end{verbatim}
Table A.13: Output produced by the code in Section A.16, except that the equations here are numbered (A.1) and (A.2) rather than (1) and (2) and the footnote is labeled \textsuperscript{a} rather than \textsuperscript{1}. Further, the page number that would appear has been deleted in this display.

A Sample Document for Your Perusal

\textbf{Author:} J. Q. Student  \\
\textbf{Date:} 16 January 2020

In this document we have used the \texttt{\setlength} commands to modify \LaTeX{}’s default page setup to make fuller use of an 8.5\"$\times$11\" page.\footnote{Note the special command \texttt{\LaTeX} provided for the display of the \LaTeX{} “logo”}. The title could have been generated using the three commands \texttt{\title{...}}, \texttt{\author{...}}, and \texttt{\date{...}} before the \texttt{\begin{document}} command—i.e., in the preamble—followed by the command \texttt{\maketitle} after the beginning of the document. The command \texttt{\verb*+␣...␣+} will cause \LaTeX{} to print whatever is between the plus signs (which could be virtually any specified character) exactly as it is. The asterisk is optional, and it makes \LaTeX{} highlight the spaces that occur inside the \texttt{verbatim} environment.

Notice that a blank line creates a new paragraph. The \texttt{\noindent} command suppresses the standard paragraph indentation. In this special space we will also demonstrate some math environment operations. Consider the equation

\[
\int_{0}^{\infty} e^{-x} \, dx = 1, \tag{A.1}
\]

where a medium sized space, specified by the command \texttt{\:}, is put between the integral sign $\int_{0}^{\infty}$, and the integrand and a small space, specified by the command \texttt{\,,}, is inserted between the integrand and the $dx$ in Eq. (A.1). The equation

\[
\frac{\partial e^{x_0 y^2}}{\partial x_0} = y^2 e^{x_0 y^2} \tag{A.2}
\]

demonstrates partial derivatives and fractions. Note the automatic generation and placement of equation numbers in the output. Also note that pages are automatically numbered, though the actual printing of page numbers can be suppressed with the commands \texttt{\pagestyle} and \texttt{\thispagestyle}.

\section*{5.} The command \texttt{\renewcommand} for changing commands that already exist. When a command already exists (as, for example, when a command is defined by the selected document class), \texttt{\newcommand} will fail. In those circumstances (as we have seen already in Section A.5), we need the command \texttt{\renewcommand}. As one particular example, the built-in command \texttt{\today} which, by default, formats the date in American style (e.g., April 3, 1938) can be changed to format the date in European style (3 April 1938) with the command

\begin{verbatim}
\renewcommand{\today}{\number\day\space \ifcase\month\or January\%
\or February\or March\or April\or May\or June\or July\or August\%
\or September\or October\or November\or December\fi \space\number\year}
\end{verbatim}

The percent signs at the end of the first two lines are not superfluous; they guarantee that \TeX{} will see this command as a single line and hence that extraneous spaces will not appear in
the output for occasional dates. Note also that this modification invokes \TeX’s case structure, which is introduced by the \texttt{\ifcase} command and terminated by the \texttt{\fi} command.

6. The command \texttt{\pagestyle{Style}}. Placed in the preamble, this command selects a particular (global) page style for the entire document. Recognized styles include \texttt{plain}, \texttt{empty}, \texttt{headings}, and \texttt{myheadings}. Details will be found in Sections 6.1.2 and C.5.3 in The \TeX{} Manual.

7. The command \texttt{\thispagestyle{Style}}. Placed at any point, this command selects a particular page style for the current page, overriding the global specification for that page alone. Recognized styles are the same as for the command \texttt{\pagestyle}. This command in the form \texttt{\thispagestyle{empty}} is commonly used to suppress page numbering on the first page of a several-page document. Details will be found in Sections 6.1.2 and C.5.3 in The \TeX{} Manual.

8. Re\TeX{}, which contains files defining the \texttt{aps} document class. These files have been created by the American Institute of Physics (AIP), the American Physical Society (APS), and the Optical Society of America (OAS) and are intended for use in preparing manuscripts for ultimate publication in the journals published by these organizations. Full documentation is contained in The REV\TeX{} Input Guide prepared by the AIP, the APS, and the OAS.\footnote{Version 4.1, which was released in final form on 11 August 2010, is compatible with \TeX{}2\epsilon. The APS website \url{publish.aps.org/revtex4} (no www) contains up-to-date information about Re\TeX{}4, links to an assortment of manuals (including one titled Re\TeX{}4.1 Author’s Guide, and a link from which that version can be downloaded. Re\TeX{} is automatically included in many standard \TeX{} distributions.}

9. The command \texttt{\input}, which simply inputs the file specified in its argument. Details are described in Section 4.4 of The \TeX{} Manual.

10. The calligraphic type style for producing upper-case calligraphic letters in math mode. This style can be invoked either with the declaration \texttt{\cal} or the command \texttt{\mathcal}. It is described in Sections 3.3.2, 3.3.8, and C.7.8 of The \TeX{} Manual.

11. Methods for placing accents over characters. For example, the command \texttt{"{o}} will produce \texttt{"{o}}, the command \texttt{\~{n}} will produce \texttt{\~{n}}, and the command \texttt{\c{c}} will produce \texttt{\c{c}}. A full listing of the possibilities will be found in Section 3.2.1 and Table 3.1 of The \TeX{} Manual.

12. The \texttt{alltt} and \texttt{verbatim} packages, which allow incorporation of computer code by reference to the actual file containing the computer program itself. On the surface, it would appear as if a file containing computer code could be incorporated into a \TeX{} document with the simple command sequence

\begin{verbatim}
\begin{verbatim} \input{FileName} \end{verbatim}
\end{verbatim}

The problem comes because the backslash that introduces the command \texttt{\input} will, in the \texttt{verbatim} environment, be treated as an ordinary character and will not be recognized as introducing a command. If, instead, we invoke the alternative \texttt{alltt} environment with the \texttt{\alltt} commands

\begin{verbatim}
\begin{alltt} \input{FileName} \end{alltt}
\end{verbatim}

the problem is solved, since \texttt{\}, \texttt{\{}, \texttt{\}}, and one or two other characters are treated specially within the \texttt{alltt} environment and the embedded command \texttt{\input} will now be properly recognized as a command to read the specified file into the \TeX{} source stream at this point. This new environment, however, will be available only if the \texttt{alltt} package is explicitly added with the command

\begin{verbatim}
\usepackage{alltt}
\end{verbatim}

placed in the preamble to the \TeX{} file.
Actually, there is at least one situation in which the `alltt` environment isn’t quite up to the task. If we want to read in a C program that specifies newline characters with \n, \TeX\ will complain that \n is an undefined command. For this case (and, of course, for the others as well), we need instead exploit the `verbatim` package (not environment), which is made available by placing the command

\usepackage{verbatim}

in the preamble. In particular, this inclusion defines a command `\verbatiminput`, invoked with a statement like

`\verbatiminput{FileName}`

When the `verbatim` package is invoked, we also have available a new environment—the `comment` environment, which can be used to “bracket” extended text that one wants to exclude from processing by \TeX.\footnote{The `verbatim` package also redefines the `verbatim` environment in ways, however, of little consequence unless the text in the environment is very extensive. If the `verbatim` package is invoked, one particularly significant change in the `verbatim` environment results in the complete ignoring of any characters following the statement `\end{verbatim}` in the same line. It is best always to place the statement `\end{verbatim}` on a line by itself.}

13. The ten characters (\$, \&, {, }, \%, \#, \_, \^, \backslash, \texttt), which have special meanings to \TeX and will not normally be printed as characters. The first seven of these characters can be printed as characters by preceding the character with a backslash, e.g., to print $\$, type $\$ in the code. The last three, however, are trickier because \^ and \backslash are themselves commands for accents over the following letter and \_ is the command for a new line. These last three characters can, however, be printed by invoking the constructions `\verb\$\^\$`, `\verb\^\$`, and `\verb\backslash\$`. Here, the `\verb` command enters verbatim mode, the immediately following + sign—any character can be used—flags the beginning of the text to be presented in verbatim mode, and the final + sign—better, repeat of the first character—flags the end of the text to be presented in verbatim mode. The backslash can also be produced in math mode with the command `$\backslash$`.

In some environments and in footnotes, the command `\verb` is forbidden. Another way to specify the printing of some of these special characters is to use the \TeX command `\char`. In its raw form, this command simply takes a number, e.g., `\char98`, which specifies the insertion of character 98 from the current font. We can, however, also specify the character by invoking the ‘tic’ operator, which instructs \TeX to calculate the numeric code from the character itself. In particular, some of the special characters that cannot be invoked directly can be invoked with the command `\char`. Thus, for example, the command `\char97` will produce the character ‘a’ while the commands `\char‘b`, `\char‘\`, `\char‘\^`, and `{\tt \char\}` will produce characters ‘b’, ‘\’, ‘\^’, and ‘\texttt’, respectively. Similarly, the command `{\tt \char‘>` will produce ‘>’, which differs in sometimes desirable ways from the character ‘>’ produced by `\$\texttt\$`.

14. The commands that allow us to create boxes as described in Sections 6.4.3 and C.13.3 in \TeX Manual. These commands include `\mbox`, `\makebox`, `\fbox`, `\framebox`, `\parbox`, and `\rule` and the `minipage` environment. All of these commands “block” the contents of the box into a structure that \TeX sees as a single unit. The commands `\fbox` and `\framebox` also place a printed rectangular box around the contents of the box created.

15. The notion of counters. In the off-the-shelf version of \TeX, each new invocation of the `enumerate` environment starts a counter off again at the beginning. Sometimes, we might wish to have the numbers in a new enumeration pick up from where the numbers in the previous one ended. We should, of course, achieve that objective in a way that does not depend on knowing explicitly what the starting number in the new environment should be. Basically, we have to invent a way to tell \TeX to remember the value at which it ended and retrieve that
value when a new environment is entered. The remembered value must be stored in a \LaTeX variable called a \textit{counter}, which must be defined with the command

\begin{verbatim}
\newcounter{hold}
\end{verbatim}

where \texttt{hold}, which cannot contain numeric characters, is the name chosen for the counter. Then, the structure

\begin{verbatim}
\begin{enumerate}
\item Text of first item.
\item Text of second item.
\setcounter{hold}{\value{enumi}}
\end{enumerate}
\end{verbatim}

... Text not in \texttt{enumerate} environment.

\begin{verbatim}
\begin{enumerate}
\setcounter{enumi}{\value{hold}}
\item Text of third item.
\item Text of fourth item.
\setcounter{hold}{\value{enumi}}
\end{enumerate}
\end{verbatim}

will achieve the desired end. In the first \texttt{enumerate} environment, the counter \texttt{enumi}, which is used behind the scenes to keep track of the number of items, is initialized to zero. It is then incremented by 1 with each new item. At the end of the environment, \texttt{enumi} stores the number of the last item. The command \texttt{\setcounter} in the first environment saves the current value of \texttt{enumi} in the counter \texttt{hold}, which survives the exit from the environment. Thus, immediately on entry to the second environment, we can use a “reflection” of the first use of \texttt{\setcounter} to restore the value that \texttt{enumi} had reached at the end of the first environment.

16. Parameters influencing placement of “floats” (figures and tables). Sometimes \LaTeX seems to have a mind of its own with regard to placement of floating objects on the current or later pages. Many of these “problems” can be cleared up by tampering with the default values of the parameters that limit the number of floats that can be put at the top or bottom of the page or, even more, by changing the parameters that stipulate the maximum fraction of a page that can be devoted to floats or the minimum amount of text that must appear on a page. These parameters are enumerated at the end of Section C9.1 in \textit{The \LaTeX Manual}. For single column presentations, the most important of these parameters are \texttt{\topfraction}, \texttt{\bottomfraction}, \texttt{\textfraction}, and \texttt{\floatpagefraction}. The command \texttt{\renewcommand} must be used to change the values of these parameters.

17. The way to change section, page, figure, table, and footnote “numbering”. By default in the \texttt{article} class, sections, subsections, subsubsections, pages, figures, and footnotes are labeled with Arabic numbers, e.g., Section 3.1.2. The format of that label as determined from the underlying counters \texttt{section}, \texttt{subsection}, \texttt{subsubsection}, \texttt{page}, \texttt{figure}, \texttt{table}, and \texttt{footnote} can be changed. If, for example, we wished the sections to be labeled with upper-case letters, the subsections with lower-case letters, and the subsubsections with arabic numbers (the default), we would simply execute the instructions

\begin{verbatim}
\renewcommand{\thesection}{\Alph{section}.}
\renewcommand{\thesubsection}{\thesection\alph{subsection}.}
\end{verbatim}

in the preamble.\footnote{Note the explicit periods and the inclusion of the section “number” in the definition of the subsection “number”.} Similarly, the commands

\begin{verbatim}
\renewcommand{\thetable}{\Roman{table}}
\renewcommand{\thepage}{\roman{page}}
\end{verbatim}
Table A.14: Structure to produce full-width text at top of double-column text.

\documentclass[twocolumn]{article}
\begin{document}
\twocolumn
[ \begin{center}
{\large\bf Title} \[12pt] % Bold title; extra space
{\large\bf Author} \[12pt] % Bold author; extra space
\end{center}
\begin{quotation}
\noindent ... Text of abstract.
\end{quotation}
]

... Text of paper
\end{document}

will change table “numbering” to upper-case Roman numbers and page “numbering” to lower-case Roman numbers. Essentially, the “numbers” on these components of a document are generated when \LaTeX executes the command \texttt{\the...}, where \texttt{...} stands for the name of the appropriate counter. Redefining this command changes the translation \LaTeX applies to the counter when generating the “number”.

18. The way to create a full-width heading and abstract above (and on the same page as) a two-column presentation of text, which exploits an optional argument to the \texttt{\twocolumn} command (not the optional argument \texttt{twocolumn} to the command \texttt{\documentclass}). One format that achieves this end uses the code is shown in Table A.14.\footnote{In more recent versions of \LaTeX, the optional argument \texttt{[12pt]} to put a bit of extra space after the title and author appears to generate error messages. The “fix” is to replace \texttt{[12pt]} with a full command, specifically \texttt{\vspace{12pt}}.} If this pattern is to work, there can be no commands that produce output between the command \texttt{\begin{document}} and the command \texttt{\twocolumn}.

19. The means to change the label in the caption of a figure. By default, \LaTeX introduces the caption of the first figure with the label ‘Figure 1:’, incrementing the number automatically with each subsequent figure. The word ‘Figure’ in this caption is the default value of the command \texttt{\figurename}, but that portion of the label can be changed with a simple command like

\texttt{\renewcommand{\figurename}{Fig.}}

in the preamble. Changing the colon after the number is harder, since the specification of that punctuation is embedded in the class file in use and is not brought to the outside in a simple command. To change the colon to a period, for example, we need to find the file \texttt{article.cls}, search through the file for colons, replace the critical ones with periods, save the file in the default directory with a new name, and then specify it rather than the standard file in the \texttt{\documentclass} command at the beginning of the \LaTeX source file.\footnote{The file \texttt{article.cls} in the version of \LaTeX installed at Lawrence University has only two colons that are not in comments; both should be changed.}
20. The means to change the label in the caption of a table. By default, \LaTeX\ introduces the caption of the first table with the label ‘Table 1:’, incrementing the number automatically with each subsequent table. The word ‘Table’ is supplied by the command \tablename\ and can be changed in the way described for \figurename\ at item 19 above. The colon can also be changed as described in the previous item, though changing the colon for figures will change it in the same way for tables.

21. The means to change the label in the title of a chapter. By default, \LaTeX\ introduces the title of the first chapter with the label ‘Chapter 1’ on a line by itself, incrementing the number automatically with each subsequent chapter. The word ‘Chapter’ is supplied by the command \chaptername\ and can be changed in the way described for \figurename\ at item 19 above.

22. The means to change the title above the table of contents and above the index. By default, \LaTeX\ labels the table of contents ‘Contents’ and the index ‘Index’. The word ‘Contents’ is supplied by the command \contentsname\ and the word ‘Index’ is supplied by the command \indexname\. Each can be changed in the way described for \figurename\ at item 19 above.

23. Using the \array\ environment to create matrices in math mode. As with the \tabular\ environment (see Section A.6), an optional argument specifies the positioning of entries in the columns, ampersands separate entries in each row, and the command \\ marks the end of each row. Thus, for example, the \LaTeX\ code

\[
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
5 & 6 & 7 & 8 \\
9 & 10 & 11 & 12
\end{array}
\]

will produce the display

\[
\begin{pmatrix}
1 & 2 & 3 & 4 \\
5 & 6 & 7 & 8 \\
9 & 10 & 11 & 12
\end{pmatrix}
\]

The entries in each cell can, of course, be much more elaborate than the simple choices made here.

24. The \LaTeX\ package \needspace, which facilitates avoiding awkward page breaks conditionally. For example, when a section title occurs close to the bottom of a page, \LaTeX\ may place the title and only a single line of the text before turning the page. Sensible styles would dictate that the section be started on a fresh page. The command \newpage\ will, of course, achieve that objective. As a document experiences further edits, however, that page break may not any longer be appropriate. Placing the command

\usepackage{needspace}

in the preamble and then, at the point where a conditional page break might be wise, inserting one of the commands

\needspace{5\baselineskip} \quad \text{or} \quad \Needspace{5\baselineskip}

will result in turning the page, but only if five or fewer lines remain at the bottom of the page. The amount of space left by \needspace\ will depend some on what penalties are in effect but will usually be close enough to be acceptable; further, \needspace\ will leave a ragged bottom even if \flushbottom\ is in effect. The command \Needspace\ will leave the requested space, will take longer to execute, and will leave a ragged bottom; the command \Needspace*\ will produce a flush bottom if \flushbottom\ is in effect. The unit of measure is the size of the length parameter \baselineskip\. The 5 in this example can, of course, be whatever number seems appropriate. Note that there is \textit{no} multiplication sign after the number.
25. The Windows batch command `cleanTEX` defined by the batch file

```plaintext
REMOVED delete all TEX intermediate files
del *.aux
del *.log
del *.dvi
del *.toc
del *.ilg
del *.idx
del *.bak
del *.sav
del *.out
del *.ind
```

and the Unix shell script defined by the file

```plaintext
#!/bin/bash
# delete all TEX intermediate files
rm -f *.sav *.out *.ind
```

which facilitate removing all \TeX/E\TeX auxiliary files\(^{77}\) in the directory in which the command or script is executed. These files can be created by direct typing or can be copied from the directory `$HEAD/tex`. The Windows file is executed by the simple statement `cleanTEX` at a command window prompt, though a path to the file may be included if the file is not located in the directory to be purged of \TeX/E\TeX auxiliary files; the UNIX file is executed by the simple statement `.cleanTEX`, though the characters `./` may be replaced by the path to the file if the file is not located in the directory to be purged of \TeX/E\TeX auxiliary files.

26. The Windows batch command `cleanbak` define by the the batch file

```plaintext
REMOVED Delete .bak files
del /s *.bak
del /s *.sav
```

and the UNIX shell script `cleanbak` defined by the file

```plaintext
#!/bin/bash
# Delete *.bak, *.sav, and *~ files
find . -name '*.bak' -exec rm {} \;
find . -name '*.sav' -exec rm {} \;
find . -name '*~' -exec rm {} \;
```

which facilitate removing all backup files\(^{78}\) in the directory in which the command or script is executed and in all subdirectories below that directory. These files can be created by direct typing or can be copied from the directory `$HEAD/tex`. The Windows file is executed by the simple statement `cleanbak` at a command window prompt, though a path to the file may be included if the file is not located in the directory at the top of the tree to be purged of backup files; the UNIX file is executed by the simple statement `.cleanbak`, though the characters `./` may be replaced by the path to the file if the file is not located in the directory at the top of the tree to be purged of backup files.

\(^{77}\)Files with file types `.aux`, `.log`, `.dvi`, `.toc`, `.ilg`, `.idx`, `.bak`, `.sav`, `.out`, and `.ind.

\(^{78}\)Files with file types `.bak` and `.sav` and files for which the last character in the name is `~.`
A.18 References

As access to the web has expanded, more and more of the information that once was printed is available more readily on the web. Searching for help on specific issues, either on the web as a whole or (perhaps more effectively) more narrowly on the \TeX Users Group site www.tug.org will yield valuable results. Further, when \TeX is installed in UNIX and UNIX-based operating systems (which includes Mac computers), the Shell command man, e.g. \texttt{man hyperref}, may bring up a page describing the indicated entity. In all full \TeX installations, the Shell command \texttt{texdoc}, e.g. \texttt{texdoc hyperref}, will bring up a description (though some of those descriptions are in languages other than English).

Numerous books have also been written for a variety of audiences and with objectives ranging from general discussions to very specific focus on particular tasks. Among the more common books are the following:


\textit{Math Into \LaTeX} (Third Edition), George Grätzer (Birkäuser, Boston) [ISBN 0-8176-4131-9 or 978-0201433111, 2000]


Additional books will likely surface in a search for \LaTeX on the Amazon or Barnes and Noble websites, though that search may also generate a number of hits for documents about rubber.

A.19 Exercises

A.1. Study carefully the \LaTeX code presented in Section A.16 and the resulting output in Table A.13, making sure you understand both the syntax of each command and the effect it produces in the output.
A.2. Use LATEX to write a letter to a friend. Format your letter so that it has

- a centered block at the top containing your name and address (center environment);
- a right-justified date (using \today and either \hfill or the flushright environment);
- a left-justified block containing an inside address (flushleft environment);
- a left-justified salutation;
- the body of the letter, containing more than one paragraph;
- a closing (e.g., Sincerely, With love, ...) positioned 3.5” from the left margin; and
- your name, spaced far enough below the closing to allow for your signature and aligned with the closing.

Suppress page numbers altogether on this letter. Include both your LATEX code and the processed output in what you submit as a solution to this exercise.

A.3. Examine one (or more) of the templates listed in Table A.6 and, in a document produced with LATEX, explain the function of each command it contains. Include both your LATEX code and the processed output in what you submit as a solution to this exercise.

A.4. The folder $\texttt{HEAD/tex}$ contains the three files radio.ps, radio.eps, and radio.pdf, each of which contains a description of a graph showing the number of nuclei of each of three species A, B, and C as a function of time as the 1000 nuclei of A initially present decay radioactively first to B and then to C. The equations describing the system are

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

and the graph shows the solution of these equations when the initial conditions are $A(0) = 1000$, $B(0) = C(0) = 0$ for the case $k_A = k_B = 0.1$. Prepare a document containing this figure, the differential equations, the initial conditions, and a brief description of the graph in English. Be sure that your text includes explicit references to the figure and the equations. You might also contrive to include a footnote somewhere. Include both your LATEX code and the processed output in what you submit as a solution to this exercise. A Crutch: The file $\texttt{HEAD/tex/sampledoc.tex}$ provides a start on the creation of a suitable source file for this exercise.

A.5. Generate LATEX code to produce the memo

\begin{center}
\textbf{MEMORANDUM}
\end{center}

\textit{Current date}

\textbf{TO:} \hspace{1cm} \textit{Insert name of recipient here.}
\textbf{FROM:} \hspace{1cm} \textit{Insert name of sender here.}
\textbf{SUBJECT:} \hspace{1cm} \textit{Insert topic here.}

\textbf{MESSAGE:}

\textit{Insert message here.}

Make sure that the date placed in the memo is the date the memo was processed (i.e., use the command \today). Include both your LATEX code and the processed output in what you submit as a solution to this exercise. \textit{Suggestion:} You may want to save this template in a file, say \texttt{memo_template.tex}, so you have it available as a starting point for the multitude of memos you will subsequently write.
A.6. In connection with a grant you received from the XYZ Foundation, you need periodically to submit a progress report. The format and content of that report are dictated by the Foundation, and the report is created by filling in the information indicated in italics in the template on page 502. Create a \LaTeX source file that you can save and use repeatedly to facilitate generating each required report, i.e., create a \LaTeX source file that, when processed, will produce the output shown below. Pay meticulous attention to the spacing in the heading and to the spacing and the rulings in the table.
PROGRESS REPORT to XYZ FOUNDATION

Due date of report

Grant Number: Grant number  Date of Award: Date of award

Title of Award: Title of award

Principal Investigator (PI): Name of PI

Investigator’s Institution: Name and address of institution

1. Please describe progress made since last report:
   Insert response.
2. Please list talks given since last report, including title and venue:
   Insert response.
3. Please give full citations for each publication since last report:
   Insert response.
4. Please describe any unanticipated difficulties encountered:
   Insert response.
5. Please identify any individuals beyond the PI who have contributed more than ten hours per week to the project during the past time period:
   Insert response.
6. Please summarize expenses since the last report:

<table>
<thead>
<tr>
<th>Category</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original grant</td>
<td>$xxx,xxx</td>
</tr>
<tr>
<td>Total expenses reported last time</td>
<td>$xxx,xxx</td>
</tr>
<tr>
<td>Available funds at start of current period</td>
<td>$xxx,xxx</td>
</tr>
<tr>
<td>Expenses in current period:</td>
<td></td>
</tr>
<tr>
<td>Salaries</td>
<td>$xxx,xxx</td>
</tr>
<tr>
<td>Fringe benefits</td>
<td>$xxx,xxx</td>
</tr>
<tr>
<td>Equipment</td>
<td>$xxx,xxx</td>
</tr>
<tr>
<td>Supplies</td>
<td>$xxx,xxx</td>
</tr>
<tr>
<td>Travel</td>
<td>$xxx,xxx</td>
</tr>
<tr>
<td>Page charges</td>
<td>$xxx,xxx</td>
</tr>
<tr>
<td>Other: Identify</td>
<td>$xxx,xxx</td>
</tr>
<tr>
<td>Total for current period</td>
<td>$xxx,xxx</td>
</tr>
</tbody>
</table>

   AVAILABLE FUNDS FOR REMAINDER OF PROJECT $xxx,xxx

Signed: ________________________________  Date: ________________________________

Typed Name of PI

Template to be reproduced in Exercise A.6.
A.A Listings

This section provides listings of the several batch files/shell scripts and the one PYTHON program involved in converting PostScript files to PDF files. Their use is described in Section A.8, and the files themselves reside in the directory $HEAD/tex.

A.A.1 ... for Windows

Batch File ceps2pdf.bat

REM ceps2pdf.bat (Convert EPS to PDF in Windows)
echo off
ps2pdf %1.eps tmp.pdf
pdfcrop tmp.pdf %1.pdf
del tmp.pdf

Batch File cps2pdf.bat

REM cps2pdf.bat (Convert PS to PDF in Windows)
echo off
ps2pdf %1.ps tmp.pdf
pdfcrop tmp.pdf %1.pdf
del tmp.pdf

Batch File rdfileeps.bat

REM rdfile.bat (Convert each file in nameonly.txt to PDF in Windows)
del dir.txt
for /f %%a in (nameonly.txt) do ( 
   ps2pdf %%a.eps tmp.pdf
   pdfcrop tmp.pdf %%a.pdf
 )
del tmp.pdf
del nameonly.txt

Batch File rdfileps.bat

REM rdfileps.bat (Convert each file in nameonly.txt to PDF in Windows)
del dir.txt
for /f %%a in (nameonly.txt) do ( 
   ps2pdf %%a.ps tmp.pdf
   pdfcrop tmp.pdf %%a.pdf
 )
del tmp.pdf
del nameonly.txt
A.A.2 … for UNIX

Shell Script ceps2pdf

#!/bin/bash
# ceps2pdf (Convert EPS to PDF in UNIX)
filename=$1
ps2pdf $filename.eps tmp.pdf
dfcrop tmp.pdf $filename.pdf
rm -f tmp.pdf

Shell Script cps2pdf

#!/bin/bash
# cps2pdf (Convert PS to PDF in UNIX)
ps2pdf $1.ps tmp.pdf
dfcrop tmp.pdf $1.pdf
rm -f tmp.pdf

Shell Script rdfileeps

#!/bin/bash
# rdfileeps (Convert each file in nameonly.txt to PDF in UNIX)
cat nameonly.txt | while read filename
do
    ps2pdf $filename.eps tmp.pdf
dfcrop tmp.pdf $filename.pdf
    rm -f tmp.pdf
done
rm -f dir.txt
rm -f nameonly.txt

Shell Script rdfileps

#!/bin/bash
# rdfileeps (Convert each file in nameonly.txt to PDF in UNIX)
cat nameonly.txt | while read filename
do
    ps2pdf $filename.ps tmp.pdf
dfcrop tmp.pdf $filename.pdf
    rm -f tmp.pdf
done
rm -f dir.txt
rm -f nameonly.txt
Python Script ExtractFileName.py

# David Cook
# ExtractFileName.py
# Strip file names in dir.txt and store results in nameonly.txt

# Open and read in existing file from execution of dir /b or ls -l
stream = open("dir.txt","r")
all_lines = stream.readlines()
stream.close()

current_line = "\n"
current_heading = ""

# Open a file for writing out
outfile = open("nameonly.txt","w")

# Loop over each line of the 'all_lines' variable,
# stripping off the file type, leaving only the
# name to be written to the output file.
for line in all_lines:
    lw = (line.split(".")[0]).strip()
    outfile.write(lw +"\n")
outfile.close()
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.

**\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 by Alan C. Hindmarsh and others. Compiled solvers, numerous example programs, and source code for the package are available for download from links at [computation.llnl.gov/casc/odepack/odepack_home.html](http://computation.llnl.gov/casc/odepack/odepack_home.html). Additional information and downloads may be found from links at [www.netlib.org/odepack](http://www.netlib.org/odepack). In particular, the two text files `opkd-sum` and `opks-sum` and 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/\LaTeX system for Macintosh computers. Information is available at the URL [www.tug.org/mactex](http://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.
- 615 Kumpf Drive
- Waterloo, Ontario
- CANADA, N2V 1K8

Voice: 800-267-6583  
FAX: 519-747-5284  
E-mail: info@maplesoft.com  
Web: www.maplesoft.com

**MATHEMATICA®**

- Wolfram Research, Inc.
- 100 Trade Center Drive
- Champaign, IL 61820-7237 USA

Voice: 217-398-0700, 800-965-3726  
FAX: 217-398-0747  
E-mail: info@wolfram.com  
Web: www.wolfram.com

**MATLAB®**

- The MathWorks, Inc.
- 3 Apple Hill Drive
- Natick, MA 01760-2098 USA

Voice: 508-647-7000  
FAX: 508-647-7101  
E-mail: info@mathworks.com  
Web: www.mathworks.com

**MiKTeX**

MiKTeX is an implementation of the TeX/LaTeX system for Windows computers. Information is available at the URL www.miktex.org. MiKTeX is also available from CTAN, the TeX/LaTeX distribution network accessible from links at www.tug.org.

**MUDPACK**

MUDPACK is a package containing many components for solving elliptic partial differential equations in two and three dimensions. The main web page for information about this package is www2.cisl.ucar.edu/resources/legacy/mudpack.

**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.
- 801 Warrenville Road
- Suite 185
- Lisle, IL 60532-4332 USA

Voice: 630-971-2337  
FAX: 630-971-2706  
E-mail: infodesk@nag.com  
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 www.gnu.org/software/octave.

**ODEPACK – see LSODE**
OzTEX
OzTEX is a shareware implementation of the TeX/LaTeX system for Macintosh computers. Information is available at the URL www.trevorrow.com/oztex. OzTEX is also available from CTAN, the TeX/LaTeX distribution network accessible from links at www.tug.org.

PYTHON/NUMPY/MATPLOTLIB/PLTLY/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/LaTeX/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/LaTeX system can be downloaded.

\TeXlive

\TeXlive is among the newer implementations of the \TeX/LaTeX 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 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 \TeX/LaTeX. Information about licensing and downloads can be found at the URL www.winedt.com. Winedt is also available from CTAN, the \TeX/LaTeX distribution network accessible from links at 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 [xemacs.sourceforge.net](http://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 [www.trilon.com](http://www.trilon.com).
Index

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

$\textsc{idlhead}$, 11, 39, 297, 412
$\textsc{maplehead}$, 136

A

aborting execution
in C, 184
in FORTRAN, 177
in IDL, 16
in MAPLE, 92
accuracy
in finding roots, 410–411
in solving ODEs, 273–274, 283, 288
with conservation laws, 274, 288
acrosread, 462
action statements, 157, 159–160
Adams/Bashforth/Moulton (ABM) methods, 275–276
adaptive methods
for integrals, 355
for ODEs, 274–275, 303, 309
adding text to graphics, see placing text in graphics
algorithms
definition, 157
for \(a^2 + b^2 = c^2\), 209
for adding, 209
for bowling, 207, 209
for controlling loops, 166
for counting, 166, 167

for exchange, 166
for finding extremes, 167, 168
for integration, 348–358
for Laplace’s equation, 168
for ODEs, 267–276
for parking lot, 207
for root finding, 408–411
for searching, 168, 209
for sorting, 209
for summing, 167
animation
in IDL, 66–68
in MAPLE, 136
arithmetic operators
in C, 181
in FORTRAN, 175
in IDL, 23
in MAPLE, 95
array indices
in C, 181
in FORTRAN, 175
in IDL, 19, 192
in MAPLE, 101
in pseudocode, 159
arrays
for data storage, 170
in C, 181
in FORTRAN, 175
in IDL, 19–25
in MAPLE, 94, 99–102
in pseudocode, 159
scalar, see scalar fields
vector, see vector fields
ASCII character codes, 5
ASCII text files, 8
aspell, 489–490
assigning values to variables
in C, 183
in FORTRAN, 175
in IDL, 18
in MAPLE, 96–97
in pseudocode, 160

B
backsolution, 438
Bessel functions, 388, 446
binary files, 8
binary number system, 4–5
Biot-Savart law, 342
bisection, see root finding
bit, 4–5
black body radiation, 149, 387
blocking statements, see also control structures
in C, 183
in IDL, 31–32, 38, 66
in MAPLE, 125
in pseudocode, 162, 172
Boolean values, see logical expressions
Boolean variables, see logical expressions
boundary value problems, 243, 254, 450, 451
byte, 5–6

C
C, 1
C commands
fclose, 192, 201, 308, 311
fgets, 206
fopen, 184, 192, 201, 206, 308, 311, 313
fprintf, 184, 192, 201, 308, 311, 313
fscanf, 206
main, 181, 191, 200
printf, 182, 183, 186, 308, 378, 380, 382, 431, 432, 434
return, 187
scanf, 186, 308, 378, 431, 432, 434
C control structures
do/while, 186, 313
for, 182–184, 191, 192, 200, 201, 206, 308, 311, 378, 380
if/else, 185, 432
switch/case/default, 185
while, 186, 431, 432
C data types
char, 181, 206
file, 184, 191, 200, 206
float, 181, 182, 191, 200, 206
int, 181, 182, 191, 200, 206
C directives, 181
#define, 181, 182, 191, 200
#include, 181, 182, 191, 200
C functions
cos, 200, 435
exp, 380, 382
pow, 181, 191, 380, 382, 431, 432
sin, 191, 200, 434, 435
sqrt, 380, 382
user-defined, 187
C header files
math.h, 181, 191, 200, 378, 380, 382, 431, 434
nr.h, 240, 241, 311, 313, 380–382, 433–435
nrutil.h, 311, 313, 433–435
stdio.h, 181, 191, 200, 378, 380, 382, 431, 434
C programs
bisect.c, 432–433, 455
decay.c, 308, 325–326
decayrk4.c, 311, 326
decayrkqs.c, 313, 327
errometertrap.c, 381–382, 390
errtrapzd.c, 380–381, 390
guessroot.c, 431
irrad.c, 190–191, 216–217
laplace.c, 182, 184
laplace_file.c, 184
pdens.c, 195, 224–225
quadpole.c, 203–204, 232–234
read_irrad.c, 236
sinrtbis.c, 433, 434
sinrtnewt.c, 435–436
trapezoidal.c, 378, 393
wavegd.c, 198–199, 228–229
C special symbols
\n (new line), 183, 184, 192, 201, 308, 311, 313, 378, 380, 382, 431, 432, 434
* (multiplication), 181
+ (addition), 181
++ (incrementing index), 206, 378, 380
- (subtraction), 181
/ (division), 181
/\*...*/ (comment), 181
%d (format), 183, 186, 192, 201, 206, 308, 380
%e (format), 183, 186, 192, 201, 206, 308, 380
%f (format), 183, 184, 186, 308, 311, 313, 380, 382, 431, 432, 434
%g (format), 206
%o (format), 380
%p (pointer), 186, 308, 313, 431, 432, 434, 436
braces, 183
cardioid, 43, 74, 384
carrying capacity, 248, 259–261
Cartesian coordinates
in MAPLE, 154
case sensitivity
in C, 181
in FORTRAN, 173
in IDL, 16
in MAPLE, 92
in pseudocode, 158
cc, see UNIX commands
center of mass
definition of, 330
of cardioid, 384
of semicircular plate, 331, 338–340
central processing unit, 3
INDEX

chaos
  Lorenz system, 320
character codes, 4–5
character strings
  in IDL, 17, 18
  in MAPLE, 94
characteristic equation, 398
charging capacitor, 82
classification
  of integrals, 329
  of ODEs, 243
color
  in IDL, see also IDL keywords: color, 54, 67, 72, 74–76
combining statements on one line
  in C, 181
  in IDL, 16
  in MAPLE, 92
  in pseudocode, 162
command files
  in IDL, see IDL batch files
  in MAPLE, see MAPLE command files
command line editing
  in IDL, 16
  in MAPLE, 103
commands
  in C, see C commands
  in FORTRAN, see FORTRAN commands
  in IDL, see IDL commands
  in MAPLE, see MAPLE commands
  in pseudocode, see pseudocommands
comments, see also documentation
  in C, 181
  in FORTRAN, 173
  in IDL, 35, 38
  in \LaTeX, 490
  in MAPLE, 127
  in pseudocode, 162–163, 166
common storage
  in C, see global variables
  in FORTRAN, 180, 303
compiler directives, 159
  in C, see C directives
  in FORTRAN, see FORTRAN directives
  in IDL, see IDL executive commands
  in pseudocode, see pseudocode directives
compiling source file, see Local Guide, 158
  in C, 184, 191, 308, 311, 313, 378, 381, 382, 431, 433, 434, 436
  in IDL, see IDL executive commands
complex numbers
  in IDL, 17
  in MAPLE, 108–109
Compton scattering, 84
conditions, see logical conditions

constants
  in IDL, see IDL special symbols
  in MAPLE, see MAPLE constants
contact information, 507–510
continuing statement on second line
  in C, 181
  in FORTRAN, 176
  in IDL, 16
  in MAPLE, 92
  in pseudocode, 162
control characters, 4
control structures
  in C, see C control structures
  in FORTRAN, see FORTRAN control structures
  in IDL, see IDL control structures
  in MAPLE, see MAPLE control structures
  in pseudocode, see pseudocode control structures
convergence criteria, 170, 273–274, 353–355, 410–411
convert .eps/.ps to .pdf, 484–485
convert .pdf to .eps/.ps, 485
Cornu spiral, 334, 365–366
coupled oscillators, 250–251, 263–264, 314, 315
CPU, see central processing unit
Cramer’s rule, 437–438
cross product
  in IDL, 80
  in MAPLE, 127–128, 155
current directory, see also default directory
curve fitting, see least squares fitting
to parabola, 153, 448
customizing
  IDL, 77
  MAPLE, 139–140
cylindrical coordinates
  in MAPLE, 155

D

data analysis, 333
data files
  decay_c.dat, 309, 325–326
  decay_f.dat, 302, 322
decayrk4_c.dat, 311, 326
decayrk4_f.dat, 304, 323
decayrkqs_c.dat, 313, 327
decayrkqs_f.dat, 305, 324
  freefall.dat, 449, 450
  irrad_c.dat, 190–191, 206, 216–217, 236
  irrad_f.dat, 189, 205, 215, 235
  irrad_idl.dat, 193, 207
  laplace_c.dat, 184–185, 211
  laplace_f.dat, 177, 210
  ldrum.dat, 66
  pdens_c.dat, 195, 224–225
  pdens_f.dat, 194, 222–223
  pdens_idl.dat, 196
quadpole_c.dat, 204, 232–234
quadpole_f.dat, 203, 230–231
quadpole_idl.dat, 204
radio.dat, 33, 35
rcdata.dat, 44
ruchardt.dat, 451
wavegd_c.dat, 199, 228–229
wavegd_f.dat, 198, 226–227
wavegd_idl.dat, 202
data types
  floating point, 6–7
  integer, 6–7
  string, 6–7
data typing
  explicit, 159
  implicit, 159
  in C, 181
  in FORTRAN, 175
  in IDL, 16
  in pseudocode, 158–159, 161
default directory, see also current directory, 9
  in IDL, 28–30, 35–37, 44, 48, 76, 290
  in MAPLE, 126, 134, 138–139
defining ODEs
  for IDL, 279–280
  for MAPLE, 117, 257, 259, 262–264, 299
deleting functions
  in MAPLE, 97
deleting variables
  in IDL, 21
  in MAPLE, 97, 111
demos
  in IDL, 16
  in MAPLE, 91, 94
diffraction patterns, see also interference patterns, 188, 389, 446, 447
  of square aperture, 48–54
dimensionless casting
  for plotting, 84
  of integrals, 331–333, 335, 342, 383, 387
  of ODEs, 120–121, 244–249, 251–253, 260, 397, 399
discretizing
  Laplace’s equation, 169
displaying data on screen
  in C, 182, 183
  in FORTRAN, 174, 176
  in IDL, 20–22
  in MAPLE, 92
  in pseudocode, 160
dividing statement between lines, see continuing statement on second line
documentation, see also comments
  in IDL pro-files, 38
  in MAPLE notebooks, 136–137
dot product
  in IDL, 25, 80
  in MAPLE, 127–128, 155
double factorial, 12
drawing space curves
  with IDL, 69–71
  with MAPLE, 140
dvipdfm, 462
dvips, 461, 479

E
electric field, 334
  in wave guide, 62–65, 196–202
  of dipole, 213
  of quadrupole, 202–204
  of relativistic particle, 84
electrostatic potential, 334
  of charged disks, 82, 83, 384
  of charged rings, 334–335, 367, 371, 388
  of finite line charge, 340–342
  of spherically symmetric distribution, 384
elliptic integrals, 153, 333, 388
e m dash, 470
e m space, 467
e n dash, 470
e n space, 467
energy levels
  in quantum oscillator, see harmonic oscillator
  in quantum well, 265–266, 401, 417–419, 422–424, 445
equilibrium points, 396, 405
equivalent resistance, 449
error function, 334, 336, 358, 362, 364, 370, 373, 374, 378, 380, 387
error messages
  in LATEX, 487–488
  improved, see improved Euler method
evaluating integrals
  with C, 377–383
  with FORTRAN, 372–377
  with IDL, 358–368
  with MAPLE, 114, 337–348, 368–371
exchange algorithm, 166
executing program, see Local Guide
exiting
  from IDL, 16
  from MAPLE, 92
expressing infinity
  in MAPLE, 338
expressions
  in MAPLE, 94–96

F
f77, see UNIX commands
Fabry-Perot interferometer, 83
factorial, 12, 104
Fibonacci numbers, 81
field mapping, 86, 87
finding eigenvalues/eigenvectors
with IDL, 39, 81
with MAPLE, 113–114, 150, 151
finding roots, see root finding
finite difference approximations
for derivatives, 209
fitting data, see curve fitting, least squares fitting
floating point number
double precision, 7
single precision, 6
fonts, see also Greek letters
in IDL, 51, 52, 68–69
formatting data files, 170–171
formatting output
in C, 183, 192, 201
in FORTRAN, 176
in IDL, 29–30
in pseudocode, 160
FORTRAN, 1
position of statements on line, 175
FORTRAN commands
close, 190, 199, 205, 302, 304
format, 179, 190, 199, 304, 305
open, 177, 190, 199, 205, 301, 304, 305
read, 178, 179, 205, 301, 373, 425, 426, 428
write, 174, 176–179, 190, 199, 301, 302,
304–306, 373, 375, 425, 426, 428
FORTRAN control structures
call, 180, 304–306, 375, 430
do while/enddo, 179, 306, 425, 426
do/enddo, 174–177, 190, 198, 199, 205, 302,
304, 373, 375
do/enddo, 180
function/return, 180, 373, 375, 376
if/then/else/endif, 178, 426
program/end, 175, 189, 198, 375, 426, 428
subroutine/return, 179, 303, 429
FORTRAN data types
caracter, 175, 205
double precision, 175
integer, 175
real, 175
FORTRAN directives
common, 180, 303, 304, 306
dimension, 174, 189, 198, 205, 303, 304, 306
external, 375, 376, 428
implicit, 175
parameter, 174, 175, 189, 198
FORTRAN functions
cos, 198, 429
exp, 373, 375, 376
sin, 189, 198, 428, 429
sqrt, 373, 375, 376
user-defined, 180, 428
FORTRAN programs
biseect.f, 426–427, 454
decay.f, 301, 322
decayrk4.f, 304, 323
decayrkqs.f, 306, 324
errqtrap.f, 375–377, 390
errtrapzd.f, 374–375, 389
guessroot.f, 425
irrad.f, 189, 215
laplace.f, 174, 177
laplace_file.f, 177
pdens.f, 194, 222–223
quadpole.f, 202–203, 230–231
read_irrad.f, 235
sinrtbis.f, 427, 428
sinrnewt.f, 429–430
trapezoidal.f, 373, 392
wavegd.f, 198, 226–227
FORTRAN special symbols
! (comments), 173
* (multiplication), 175
** (exponentiation), 175, 189, 373, 375, 376,
425, 426
+ (addition), 175
- (subtraction), 175
/ (division), 175
A (in format statements), 179, 301, 373, 376,
425, 426, 428
C (comments), 173
E (in format statements), 176
F (in format statements), 176, 177, 301, 304,
305, 373, 375, 428
I (in format statements), 176, 375
FORTRAN subroutines, 179–180
user-defined, 429
four-slit interference pattern, 84
Fourier series, 337, 347–348, 385
Fraunhofer diffraction, 188–193
Fresnel integrals, see also Cornu spiral, 334, 389
functions
in C, see C functions
in FORTRAN, see FORTRAN functions
in IDL, see IDL functions
in MAPLE, see MAPLE functions
typical available, 160
G
gamma function, 12–13
Gaussian distribution, 333, 387
Gaussian elimination, 438–439
Gaussian integration, see quadrature
gedit, 1
ghostview, 462
global positioning, 448
global truncation error, see truncation error,
global
global variables
in C, 187, 310
in FORTRAN, see common storage
in IDL, see common storage
golden ratio, 444
gravitational force, 252
Greek letters, see also fonts
<table>
<thead>
<tr>
<th>INDEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>in IDL, 51, 68–69</td>
</tr>
<tr>
<td>in \LaTeX, 470–471</td>
</tr>
<tr>
<td>in MAPLE, 95</td>
</tr>
</tbody>
</table>

**H**

hard copy
- from IDL, 47–48
- from MAPLE, 132–135


hazards of integer arithmetic, 17

Helmholtz coil, 342–346

help, see on-line help

hexadecimal number system, 5

high-pass filter, 44–47

high-resolution graphing
- in IDL, 26–27, 40–65
- in MAPLE, 128–132

Hooke's law, 397

hyperlinks, 483–484

**I**

IDL, 1

idl, 15

IDL batch files, 35–36

IDL command files, see IDL batch files

IDL commands, see also IDL functions, IDL procedures
- axis, 43, 74, 419
- bindgen, 22
- broyden, 443
- bytarr, 20, 66
- byte, 18, 23
- call_function, 36
- cd, 76–77
- cindgen, 22
- close, 28–30, 33, 35, 44, 66, 177, 184
- complex, 18, 23
- complexarr, 20
- contour, 49, 52–53, 58, 65
- cramer, 440
- dblrarr, 20
- dblcomplexarr, 20
- dcindgen, 22
- dcomplex, 18
- delvar, 21
- demo, 16
- device, 48, 53, 74, 76, 177, 184
- dindgen, 22
- doc_library, 35, 38
- double, 18
- eigenql, 40
- eigenvvec, 40
- elmhes, 40
- erase, 70, 71
- errplot, 45
- exit, 16
- extract_slice, 61
- findgen, 22, 26, 36, 41, 43, 44, 46, 58, 68–70, 73, 74, 76, 283, 286, 292, 359–361, 364, 366, 368, 414–416
- fix, 18
- float, 18, 74, 359–361, 413, 414
- flow3, 65, 88
- fltarr, 20, 21, 31–33, 35, 44, 58, 66, 68, 177, 184, 188, 201, 276, 283, 286, 292, 360, 361, 364, 366, 368, 414, 416, 417
- for matrix multiplication, 24
- fx_root, 411–413
- fz_roots, 411, 412, 414
- general structure of, 19
- help, 20–21
- hqr, 40
- indgen, 22
- intarr, 20, 21
- invert, 25, 26
- isosurface, 60
- journal, 30–31, 47
- lindgen, 22
- loadct, 74, 177, 184
- lonarr, 20
- long, 18
- lsode, 280–281, 284, 286–287
- lubisect, 413, 418
- ludeigen, 39–40
- ludec, 80, 440
- ludifeq, 45, 281–282, 287–288
- luqsimp, 362–368
- lusol, 80, 440
- max, 70, 416
- min, 70
- movie, 67
- newton, 443
- newton, 412–413, 416
- obj_new, 61
- openr, 29–30, 33, 35, 44, 66, 177, 184
- openw, 28–29
- plot_field, 64
- plot_io, 45
- plot_oil, 45
- plot oo, 45
- plotradio, 35
- plotrc, 47
- plots, 47, 70
- polar contour, 55–56
- polar_surface, 55–56
IDL keywords, 19, 42, 45, 52

tol, 280
ax, 51, 71
az, 51, 71
c_labels, 53
c_linestyle, 53, 65
close, 48, 74, 76
color, 63, 74, 76
decomposed, 53, 74, 177, 184
double, 416
filename, 28, 48, 74, 76, 78
format, 29–30
free, 66, 72
get_lun, 29
grid, 56
inches, 74
init, 281, 283, 285, 287, 288
isotropic, 56, 58, 73
length, 63, 64
levels, 53, 65
linestyle, 41, 42, 46, 47, 277, 283, 285, 288
maxsteps, 281
n, 64
nlevels, 52, 56, 58
nodata, 76, 288, 419
nsteps, 64
nvecs, 63
nx, 49–50, 55, 57, 63, 68, 192, 195, 201, 204
ny, 49–50, 55, 57, 63, 68, 192, 195, 201, 204
nz, 57, 195, 204
overplot, 65
polar, 43, 74
polygons, 61
prompt, 79
psym, 47
rtol, 280
set, 67
showload, 67
subtitle, 43, 74, 285
system_variables, 78
t0, 281, 283, 285, 287, 288
t3d, 60, 70, 71
tf, 281, 283, 285, 287, 288
ticklen, 27, 41, 69, 414, 416
title, 27, 41, 43, 45, 51, 52, 63, 66, 69, 72, 277, 283, 285, 288, 415
tol, 281, 285
tolf, 416
tolx, 416
verbose, 28
xaxis, 43, 74
xlog, 45
xoffset, 74
xpos, 72

xrange, 49–50, 55, 57, 60, 63, 68, 70, 192, 195, 201, 204, 415
xsize, 66, 72–74
xstyle, 34, 35, 41, 43, 74, 419
xtitle, 33, 41, 45, 51–53, 58, 63, 65, 69, 277, 283, 285, 288, 415, 416
yaxis, 43, 74, 419
ylog, 45
yoffset, 74
 ypos, 72
yrange, 45, 49–50, 55, 57, 60, 63, 68, 70, 76, 192, 195, 201, 204, 294, 296, 414, 415
ysize, 66, 72–74
ystyle, 34, 35, 41, 43, 45, 74, 419
ytitle, 33, 41, 45, 51–53, 58, 63, 65, 69, 277, 283, 285, 414, 416, 419
zrange, 52, 57, 60, 66, 195, 204
ztitle, 51, 52

IDL linestyle codes, 42
IDL pro-files, see IDL batch files, IDL functions, IDL procedures
IDL procedures, see also IDL commands, IDL functions, IDL programs, 19, 35–36
user-defined, 36
IDL programs, see IDL commands, IDL functions, IDL procedures

IDL special symbols
!3 (simplex Roman; default), 51, 52, 69, 415, 416
!4 (simplex Greek), 51, 52, 415, 416
!6 (complex Roman), 69
!8 (complex Italic), 69
!C (new line), 51, 52
!D (subscript level), 419
!E (exponent level), 69
!N (normal level), 69, 419
!pi, 192, 195
"..." (enclose strings), 18
'...' (enclose strings), 18
* (multiplication), 23
+ (addition), 23
- (subtraction), 23
/ (division), 23
: (range), 22
; (comments), 35
:+;= (documentation), 38
= (assignment), 18
? (help), 34
@ (execute command file), 36, 38
$ (continue statement), 16, 27, 38, 41, 43, 48, 50, 51, 53, 56, 57, 60–61, 63, 66
& (separate statements), 32
^ (exponentiation), 23
Inf (∞), 50, 79
NaN (not a number), 79, 418
IDL symbol codes, 46
IDL system variables, 71–72, 74
!p.background, 71
INDEX

\pcharsize, 71, 415
\pcolor, 72
\plinestyle, 72
\pmulti, 44, 71, 285, 415
\psym, 72
\pwidth, 72
\pheight, 72
\pi (\pi), 23, 43, 46, 50, 55, 63, 68, 74,
359–363, 365, 368, 415, 416
\prompt, 72
\xmargin, 73, 74
\xrange, 72
\xstyle, 72
\yrange, 73, 74
\yrange, 72, 292
\ystyle, 72
\zrange, 72
\zstyle, 72

idlde, 15
improved Euler method, 269–270, 278
indices in \LaTeX, 481
indicial equation, 266
inert commands
in MAPLE, see suppressing evaluation in
MAPLE
inhomogeneities in linear equations, 437
initial value problems, 243
initialization files
in IDL, 77
in MAPLE, 139–140
integer
signed, 6
unsigned, 6
interference patterns, see also diffraction
patterns
Fabry-Perot, 83
from four slits, 84
internal storage of expressions
in MAPLE, 96
interpolation points, 351, 352
interpreting source code, see also Local Guide,
158
interrupt
in IDL, 16
in MAPLE, 92
ispell, 489–490

K
keywords
in IDL, see IDL keywords
Kirchhoff’s laws, 449
Knuth, Donald, 457
Kronecker delta, 336

L
Lagrange equations
for 2D harmonic oscillator, 151
for compound pendulum, 152
Laguerre polynomials, 149
Laguerre’s method, see root finding
Lamport, Leslie, 457
Laplace transforms, 13–14, 255–256, 316
definition, 13, 117, 255
for ODEs, 255
in MAPLE, 117, 153
Laplace’s equation, 451
in C, 181–184, 211
in FORTRAN, 173–177, 210
in IDL, 188
in pseudocode, 168–170
\LaTeX, 1
accents in, 493
counters in, 494
dashes in, 469
math mode in, 470–471
quotation marks in, 470
special characters in, 494
latex, 460, 461, 479
\LaTeX commands
\textbackslash \textit{} (italic correction), 466
\textbackslash \textbackslash{}  (new line), 468, 473
\textbackslash{}{} (negative thin space), 471
\textbackslash{}{} (thin space), 471
\textbackslash{}{} (allow hyphen), 470
\textbackslash{}{} (medium space), 471
\textbackslash{}{}{} (thick space), 471
\addcontentsline, 480–481
\addtocontents, 480–481
\Alph, 472, 495
\alph, 472, 495
\and, 486
\arabic, 472
\author, 490
\bf, 490
\bfseries, 466
\cal, 493
\caption, 473, 474, 476, 479
\centerline, 476
\chapter, 470
\chaptername, 497
\char, 494
\clearpage, 467
\contentsname, 497
\date, 490
\definemathcolor, 483, 487
\documentclass, 458, 460, 463, 484, 490
\documentstyle, 458
\fbox, 494
\figurename, 496
\footnote, 470, 490
\footnotesize, 467
\frac, 490
\framebox, 494
\hspace, 467, 468
INDEX 521

LaTeX declarations, see \LaTeX{} commands

LaTeX{} document classes, 459

\texttt{article}, 463, 470
\texttt{book}, 463, 470
\texttt{report}, 463
\texttt{slides}, 463

LaTeX{} environments, 468–469

\texttt{\ldots{}} (in-line equation), 470
\texttt{\ldots{}} (displayed equation), 470
\texttt{$\ldots{}$} (in-line equation), 470
\texttt{array}, 497
\texttt{center}, 468, 473, 475, 479, 490
\texttt{comment}, 494
\texttt{displaymath}, 470
\texttt{document}, 459, 460, 490
\texttt{enumerate}, 471–472, 494, 495
\texttt{eqnarray}, 470
\texttt{equation}, 470, 490
\texttt{figure}, 474–477, 479
\texttt{flushleft}, 490
\texttt{itemize}, 471–472
\texttt{minipage}, 494
\texttt{pict2e}, 480
\texttt{picture}, 474, 480
\texttt{quotation}, 468
\texttt{quote}, 468
\texttt{table}, 472–474
\texttt{tabular}, 472–474
\texttt{tikzpicture}, 477–480

LaTeX{} length parameters, 464

\texttt{\baselineskip}, 467, 469
\texttt{\baselinestretch}, 469
\texttt{\bottomfraction}, 495
\texttt{\evensidemargin}, 463, 482
\texttt{\floatpagefraction}, 495
\texttt{\itemsep}, 472
\texttt{\marginparwidth}, 482
\texttt{\oddsidemargin}, 463, 490
\texttt{\parindent}, 463, 490
\texttt{\parsep}, 472
\texttt{\parskip}, 463, 472, 490
\texttt{\textfraction}, 495
\texttt{\textheight}, 463, 490
\texttt{\textwidth}, 463, 482, 490
\texttt{\topfraction}, 495
\texttt{\topmargin}, 463, 490

LaTeX{} \textit{Manual}, 457

LaTeX{} output

from MAPLE, 135–136

LaTeX{} packages, 469

\texttt{alltt}, 493
\texttt{color}, 483, 487
\texttt{graphicx}, 474–477, 483–484
\texttt{hyperref}, 483–484, 487
\texttt{ifthen}, 486–487
\texttt{makeidx}, 481–482
\texttt{maple2e}, 136
\texttt{maplestd2e}, 136
\texttt{needspace}, 497
\texttt{showidx}, 482
\texttt{tikz}, 474, 477–480
\texttt{verbatim}, 493–494

LaTeX{} page previewers, see also \texttt{xdi}, \texttt{yap}, 488–489

LaTeX{} preamble, 463–465

LaTeX{} sample, 459–462, 490–491

LaTeX{} special symbols

\texttt{''} (closing double quotation), 470
\texttt{---} (em dash), 469
\texttt{--} (en dash), 469
\texttt{-} (minus sign), 469
\texttt{\textasciigrave} (hard space), 469
\texttt{''} (opening double quotation), 470

LaTeX{} templates, 464

LaTeX{} type size, 466

LaTeX{} type styles, 465–466

lawn mower, 443

least squares fitting, 449–451

Legendre polynomials, 147, 358, 385, 445

Lennard-Jones potential, 447

licensing of software, 14

linking object code, see \textit{Local Guide}, 158

lists

in MAPLE, 94, 96, 99

Local Guide, 2, 8, 9, 11, 14, 15, 33, 35, 48, 78, 91,
92, 128, 140, 173, 177, 181, 184, 189,
191, 192, 198, 199, 203, 204, 237, 239,
240, 276, 297, 300, 307, 342, 358, 372,
373, 377, 378, 380, 411, 424, 425, 430,

logical conditions, see \texttt{logical} expressions

\texttt{logical} expressions

in C, 183

in FORTRAN, 175

in IDL, 19, 32–33

in \LaTeX{}, 486–487

in MAPLE, 125–126

in pseudocode, 160–161

logical operators, see \texttt{logical} expressions

logical unit number (LUN)

in IDL, 29

logistic growth, 248, 259–261

long integer

signed, 6

unsigned, 6

loops

in C, 183

in FORTRAN, 175

in IDL, 31–33

in MAPLE, 105, 109, 110, 124–126

in pseudocode, 163–164

Lorentz distribution, 82, 383, 387

Lorentz lineshape, 37

---
INDEX

1p, see UNIX commands
LSODE contact information, 507
LU decomposition, 439

M

machine code, 158
Macintosh OS, 1
MacTeX
   contact information, 507
magnetic field, 334
   in wave guide, 62–65, 196–202
   of circular loops, 41, 48, 85, 212, 342–346, 388
   of Helmholtz coil, 342–346
   of solenoid, 84
magnetic vector potential, 334
MAPLE, 1
maple, 92
MAPLE command files, 126–128
   crossdot, 127, 155
   testmultiply, 127
MAPLE commands, see also MAPLE functions, 94
   about, 118, 140–141
   additionally, 118, 140–141
   alias, 144
   animate, 136
   appendto, 133
   argument, 108
   Array, 100, 101, 109, 347, 370
   array, 371, 421–423
   assigned, 97
   assume, 114, 118, 121, 140–141, 259, 339, 341, 342, 347, 349, 407
   asympt, 116
   collect, 107, 260
   Column, 114
   combine, 109, 342
   conjugate, 108
   contourplot, 131
   convert, 102, 106, 109, 113, 115, 142, 153, 342, 345, 346, 370, 371, 407, 408, 421, 423
   cos, 340
   cross, 127–128
   CrossProduct, 342
   Curl, 122, 123, 154, 155
   currentdir, 138–139
   dchange, 121
   denom, 107, 260
   Determinant, 112, 264
   Diff, 116, 118, 120, 257, 259, 262–264, 299
   diff, 114, 116, 118, 129, 147, 150, 258, 405
   display, 131
   Divergence, 122, 123, 154, 155
   done, 92
dot, 127–128
   DotProduct, 342
dsolve, 118–121, 256–267, 298–300
   Eigenvectors, 113
eval, 106, 110, 118, 119, 258–261, 264, 265, 300, 341, 345–346, 405, 408, 421
evalc, 108, 109
evalf, 104, 111, 338, 346, 347, 368–371, 404, 405, 421
   expand, 105–107, 109, 110, 121, 142, 146, 346, 407, 420
   factor, 105, 118, 121, 146
   fsolve, 345, 419–424, 440, 442
   GenerateMatrix, 264
   GetCoordinates, 122
   Gradient, 122, 154, 155
   Im, 108
   infolevel, 120
   Int, 116, 369–371
   int, 114, 337–348, 368–371
   interface, 127, 133, 143, 300, 339, 341, 342, 347, 407
   invlaplace, 117, 153
   is, 258, 259
   laplace, 117, 153
   Laplacian, 122, 123, 154, 155
   latex, 135–136
   lhs, 258
   Limit, 116
   limit, 116, 259, 262, 339, 407
   listplot, 370, 371, 421
   Matrix, 101, 112, 113
   MatrixAdd, 340
   MatrixInverse, 112
   mtaylor, 115
   mtaylor, 115
   Multiply, 112
   mtaylor, 115
   Normalize, 113, 114
   numer, 260
   odeplot, 299
   op, 105–107, 115, 146, 345
   piecewise, 144
   plot, 129, 130, 144, 260–262, 344, 349, 406, 424
   plot3d, 131
   plotsetup, 132–134
   polar, 108
   print, 109, 128, 143–144
   protect, 98
   quit, 92
   Re, 108
   read, 126, 127
   restart, 92, 103, 120, 143, 259
   rhs, 118, 129, 258, 260, 262, 264
   root_by_bisection, 422
   ScalarMultiply, 338, 343, 347
   series, 116, 144, 259, 342, 407
   SetCoordinates, 122
simplify, 108, 109, 114, 117, 119, 142, 146,
258, 259, 262, 340, 342–344, 349
solve, 110, 111, 119, 258, 264, 339, 345,
403–407, 440, 442
spacecurve, 140
stop, 92
subs, 106, 119–121, 146, 260, 339, 343, 346, 349
subsop, 107, 146
Sum, 144, 349
sum, 144
taylor, 115, 339, 341, 344, 346, 347, 407,
408
textplot, 131
Trace, 112
Transpose, 112
unapply, 143
unprotect, 98, 129
value, 116, 258, 259
Vector, 99, 101, 338
VectorField, 122, 123
whattype, 106, 115
with, 111, 112, 117, 124, 131, 338, 340
writeto, 133

MAPLE constants, 95
false, 95
I (√−1), 95, 108
infinity (∞), 95, 114, 116, 338, 339
Pi (π), 95, 97, 104, 109, 131
true, 95

MAPLE control structures
for do/end do, 99, 100, 105, 109, 110, 347,
370, 371, 421–423
for/from/by/while/do/end do, 124–125
if/then/elif/else/end if, 126

MAPLE environment variables, 94
_ENV*, 94
_ENVAllSolutions, 264, 404
_ENVDropMultiplicity, 404
_ENVExplicit, 403
_MaxSols, 404
Digits, 94, 104, 111, 369, 404
infolevel, 257, 262
Order, 94, 115, 266

MAPLE functions, see also MAPLE commands,
94–98
abs, 108
cos, 95–97, 109, 128, 140, 342, 347, 349,
406, 420
cot, 423
exp, 108, 114, 115, 117, 128, 370, 406, 420
factorial, 104
RootOf, 403
sin, 97, 108, 109, 116, 117, 128, 131, 140,
337, 338, 340, 342, 347, 349, 406, 420,
422
sqrt, 104, 113, 115, 339, 340, 346, 371, 405,
423

MAPLE keywords, 95
abserr, 300
assume, 119, 340
axes, 140
axesfont, 129, 130, 260–262, 344, 349, 370,
371, 406, 421, 424
color, 129–131, 140, 260–262, 299, 344,
349, 370, 371, 406, 421, 424
conjugate, 340
continuous, 338
contours, 131
digits, 369
exp, 109
font, 131, 140
grid, 131
gridlines, 424
labelfont, 129, 130, 260–262, 344,
349, 370, 371, 406, 421, 424
labels, 129, 130, 260–262, 344, 349, 370,
371, 406, 421, 424
linestyle, 129, 130, 260–262
method, 299, 369–371
numpoints, 129, 130, 140
orthogonal, 340
output, 300
parfrac, 106, 153
plotoptions, 133
plotoutput, 133
reerr, 300
scaling, 130, 131
showassumed, 341, 342, 347
thickness, 129–131, 140, 260–262, 299, 344,
349, 370, 371, 406, 421, 424
tickmarks, 424
title, 129, 130, 260, 261
titlefont, 129, 130, 260, 261
trig, 109
type, 299, 300
verboseproc, 300
view, 140, 424

MAPLE logical expressions, see logical
expressions in MAPLE

MAPLE manuals, 91

MAPLE notebooks, 136–137

MAPLE packages, 111–112, 124
DEtools, 116, 121
DotProduct, 340
inttrans, 94, 117
LinearAlgebra, 111, 128, 264, 338, 340,
342, 343, 347
orthopoly, 148, 150
PDEtools, 116, 120
plots, 130–132, 299, 370, 371

MAPLE notebooks, 136–137

MAPLE packages, 111–112, 124
DEtools, 116, 121
DotProduct, 340
inttrans, 94, 117
LinearAlgebra, 111, 128, 264, 338, 340,
342, 343, 347
orthopoly, 148, 150
PDEtools, 116, 120
plots, 130–132, 299, 370, 371
student, 116
VectorCalculus, 122

MAPLE special symbols
! (escape to OS), 93
! (factorial), 104
* (multiplication), 95, 104, 113
** (exponentiation), 95
+ (addition), 95, 104
, (separate items in column), 101
- (subtraction), 95
-> (function definition), 95, 97, 109, 116,
144, 349, 423
.. (dimension array), 99–101
/ (division), 95, 104
:: (specify data type), 127
:= (assignment), 95–101, 109, 116
# (comments), 127
$ (sequence operator), 114, 118
% (previous statement), 105, 109, 111
%% (second previous statement), 105
%% (third previous statement), 105
_C1 (ODE constant), 118, 256, 258, 264, 265
_C2 (ODE constant), 118, 256, 258, 264, 265
_Z (generic variable), 403
_Z1 (arbitrary integer), 265
^ (exponentiation), 95, 97, 104, 105
angle brackets (<...>) (create array), 101,
340, 342, 347
P1 (π), 421
vertical bar (separate columns), 101, 340,
342, 347

MAPLE statements
termination of, 92

MAPLE system variables
libname, 139

MATHEMATICA, 1

MATHEMATICA contact information, 508

mathematical functions
in C, see C functions
in FORTRAN, see FORTRAN functions
in IDL, see also IDL functions, IDL
procedures, IDL programs, 23
in MAPLE, see MAPLE functions, MAPLE
programs, MAPLE commands

mathematical symbols
in IDL, 68–69

MATLAB, 1

MATLAB contact information, 508

matrices
in IDL, 20–25
in MAPLE, 99–102

matrix multiplication
in IDL, 24
in MAPLE, 112–113, 153

max-min problems, 396

MAXIMA, 1

MAXIMA contact information, 507
Maxwell-Boltzmann distribution, 386
method of bisection, see root finding

method of Frobenius, 266–267
method of least squares, see least squares fitting
midpoint method, 272
midpoint rule, see quadrature

MiKTeX, 1
MiKTeX contact information, 508
ML (left mouse button), 9
MM (middle mouse button), 9

moment of inertia
definition of, 331
of cardioid, 384
of semicircular plate, 331, 340, 363

motion
do membrane, 66–68, 88
do string, 88

MR (right mouse button), 9

MUDPACK contact information, 508

Müller’s method, see root finding
multi-way selection
in C, 185–186
in FORTRAN, 178
in pseudocode, 163

multistep methods
for ODEs, 275–276

NAG library contact information, 508

named common
in C, see common variables
in FORTRAN, see common storage
in IDL, see common storage

natural frequencies
of spherical cavity, 445
of stiff bar, 445
of three-mass system, 444
of two-mass system, 397–398, 405–406,
413–414

Newton's method, see root finding
Newton's second law, 329, 330, 397, 399

normal distribution, see Gaussian distribution

normal modes
of membrane, 48–54, 85
of string, 88

numerical integration, see quadrature
Numerical Recipes, 237–242, 362, 363
for ODEs
algorithm routines, 302, 309
driver routines, 303, 309
stepper routines, 303, 309
in C, 309–314, 433–436
rtbis.c, 433–436
rtnewt.c, 433, 435–436
rtsafe.c, 433
zbrak.c, 436
broydn.c, 443
bstep.c, 309
caldat.c, 240
flmoon.c, 240–241
<table>
<thead>
<tr>
<th>Function</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>gaussj.c</td>
<td>440</td>
</tr>
<tr>
<td>julday.c</td>
<td>240</td>
</tr>
<tr>
<td>lubksb.c</td>
<td>440</td>
</tr>
<tr>
<td>ludcmp.c</td>
<td>440</td>
</tr>
<tr>
<td>mmid.c</td>
<td>309</td>
</tr>
<tr>
<td>mnewt.c</td>
<td>443</td>
</tr>
<tr>
<td>newt.c</td>
<td>443</td>
</tr>
<tr>
<td>nrutil.c</td>
<td>240, 311, 313, 382, 433, 435</td>
</tr>
<tr>
<td>qromb.c</td>
<td>382, 390</td>
</tr>
<tr>
<td>qsimp.c</td>
<td>382, 390</td>
</tr>
<tr>
<td>qtrap.c</td>
<td>381–382</td>
</tr>
<tr>
<td>rk4.c</td>
<td>309–311</td>
</tr>
<tr>
<td>rkck.c</td>
<td>309</td>
</tr>
<tr>
<td>rkqs.c</td>
<td>309, 312–314</td>
</tr>
<tr>
<td>rtsafe.c</td>
<td>436</td>
</tr>
<tr>
<td>svbksb.c</td>
<td>440</td>
</tr>
<tr>
<td>svdcmp.c</td>
<td>440</td>
</tr>
<tr>
<td>trapzd.c</td>
<td>379–382</td>
</tr>
<tr>
<td>tridag.f</td>
<td>440</td>
</tr>
</tbody>
</table>

in FORTRAN, 302–307, 427–430

rtbis.f, 427–430
rtnewt.f, 427–430
rtsafe.f, 427
zbrak.f, 430
broydn.f, 443
bsstep.f, 303
caldat.f, 239
flmoon.f, 238–239
gaussj.f, 440
julday.f, 239
lubksb.f, 440
ludcmp.f, 440
mmid.f, 302
mnewt.f, 443
newt.f, 443
qromb.f, 377, 389
qsimp.f, 377, 389
qtrap.f, 375–377
rk4.f, 302–304
rkck.f, 302
rkqs.f, 303, 305–307
rtsafe.f, 430
svbksb.f, 440
svdcmp.f, 440
trapzd.f, 374–377
tridag.f, 440

NUMERICAL RECIPES contact information, 508

nybble, 5

O

object code, 158
object-image relations, 83
octal number system, 5
OCTAVE, 1
OCTAVE contact information, 508

on-line help
in IDL, 34–35

in MAPLE, 94

ordinary differential equations
with C, 307–314
with FORTRAN, 300–307
with IDL, 276–298
with MAPLE, 117–121, 256–267, 298–300
orthogonal functions, 336–337

OzTEX contact information, 509

P

page previewer
in \LaTeX, 488–489

partial fractions, 106, 153, 315

passwords, see Local Guide

path (in directory tree), 8

path specification
in IDL, 76
in MAPLE, 138

PDF files, 461–462

ps2pdf, 462, 479, 484–485

pseudocode comments, 162–163, 166

pseudocode control structures, 157, 162–165

Planck radiation law, 84

polar coordinates
in IDL, 42–43, 55–56
in MAPLE, 108–109, 122–123

population biology, 248, 259–261, 321

PostScript, 48

preallocation of arrays
in IDL, 20

predictor-corrector methods, 270

pro-files, see IDL batch files, IDL functions, IDL procedures

procedure, 165
program, 157

projectile motion, see trajectories

properties
in IDL, see IDL keywords
proprietary software, 14

protected variables
in IDL, see IDL special symbols, IDL system variables
in MAPLE, 98

ps2pdf, 462, 479, 484–485

pseudocode comments, 162–163, 166

pseudocode control structures, 157, 162–165

CASE-END_CASE, 163

EXECUTE, 164
FOR-THRU-DO, 164, 170–172
FOR-THRU-DO-STEP, 164
IF-THEN-ELSE-END_IF, 3, 4, 162, 167, 168
LOOP-END_LOOP, 3, 4, 163, 166–168
REPEAT-UNTIL, 164
WHILE-DO, 164
pseudocode directives
DIMENSION, 159, 170, 172
pseudocommands
ATTACH, 172
CLOSE, 172
READ, 160
WRITE, 160, 170, 172
PYTHON, 1
PYTHON scripts
ExtractFileName.py, 505

Q
Q of resonant circuit, 85
quadratic formula, 110, 402, 403, 443, 444
quadrature, 348–358
  adaptive algorithms, 355
  assessing error in, 353–355, 358–361,
    372–374, 378–379
C programs, 377–383
FORTRAN programs, 372–377
Gauss-Legendre, 358
Gaussian, 355–358, 360–361, 386, 389
  iterative algorithms, 355
  midpoint rule, 153, 349
Newton-Cotes, 349–355
recursive trapezoidal rule, 352
Romberg integration, 353
roundoff error, see roundoff error
Simpson’s rule, 153, 351, 353, 357, 360, 386, 389
trapezoidal rule, 350, 359, 372, 378
truncation error, see truncation error
with IDL, 358–368
with MAPLE, 368–371
quality factor, see Q of resonant circuit
quantum energy levels, see energy levels
quantum probabilities, 336
  in hydrogen atom, 56, 86, 194–196, 213, 384, 388
  of harmonic oscillator, 336, 363, 370, 386
quantum scattering, 148
quantum wave function, see wave function

R
radius of gyration
  definition of, 331
  of cardioid, 384
  of semicircular plate, 340
railroad problem, 446
range of projectiles, see trajectories
Rayleigh criterion, 447
reaction kinetics, 320
reading data
  from files
    in C, 206
    in FORTRAN, 205
  from keyboard
    in C, 186
    in FORTRAN, 178–179
    in IDL, 78–79
    in pseudocode, 160
recasting
  algorithms, 209
  control structures, 207
recursion relationship, 266
redirection of output in UNIX, 177, 184
reflection at dielectric film, 83
relativity
  mass/momentum/energy, 83
  motion under constant force, 338
relaxation, 169–170
residual, 410
resonance
  in LRC circuit, 85, 250, 316
restoring initial state
  in IDL, 78
  in MAPLE, 92, 140
returning to command level
  in IDL, 16
  in MAPLE, 92
returning to operating system
  from IDL, 16
  from MAPLE, 92
ReVTeX, 493
Richardson extrapolation, 352, 360
Rodrigues’ formula, 147
Romberg integration, see quadrature
root finding
  accuracy in, 410–411
  by bisection, 408, 412, 426–428, 432–435
  by Laguerre’s method, 410, 411
  by Müller’s method, 410, 411
  by Newton’s method, 408–409, 412,
  by secant method, 409
  by trial and error, 424–426, 431–432,
    441–442
  in C, 433–436, 443
  in FORTRAN, 427–430, 443
  in IDL, 411–419, 442
  in MAPLE, 403–408, 419–424, 442
roundoff error
  in numerical quadrature, 353
  in ODEs, 273
rubber lengths in \LaTeX, 463
Ruchardt’s experiment, 451
INDEX

Runge-Kutta methods, 270–273
running program, see Local Guide

S

saving files, see writing data to files
scalar fields
  three-dimensional, 194–196
    in C, 195
    in FORTRAN, 194
    in IDL, 195–196
  two-dimensional, 188–193
    in C, 190–191
    in FORTRAN, 189
    in IDL, 192–193
Schrödinger equation, 254–255, 265–266, 293–297, 401, 417–419
search path
  in IDL, 33, 35–37, 77
  in MAPLE, 139
secant method, see root finding
separating statements on one line
  in C, 181
  in IDL, 16, 32
  in MAPLE, 92
  in pseudocode, 162
sequences
  in MAPLE, 94, 98–99
series solutions to ODEs, 266–267
sets
  in MAPLE, 94, 99
shooting method, 290
sigmoid curve, 248, 262
simple pendulum, 319, 332, 346–347, 388
Simpson’s rule, see quadrature
simultaneous equations
  by direct methods, 439
  by iterative methods, 439
  linear, 437–440
  nonlinear, 440–443
singular equations, 438
singular value decomposition, 440
solving Laplace’s equation, see Laplace’s equation
solving linear equations, 437–440
  in IDL, 25–26
  in MAPLE, 111
solving ODEs
  with C, 307–314
  with FORTRAN, 300–307
  with IDL, 276–298
  with MAPLE, 256–267, 298–300
source code, 158
spawning to operating system
  from IDL, 16
special symbols
  in C, see C special symbols
  in FORTRAN, see FORTRAN special symbols
in IDL, see IDL special symbols
in MAPLE, see MAPLE special symbols
spherical coordinates
  in MAPLE, 155
spurious roots, 395, 401, 418, 423
stable equilibrium, 396
standing waves
  in cube, 86, 213
  in string, 253–254, 264–265, 290–292, 318
Stark effect, 81, 151, 444
startup files, see initialization files
structure of data files, 170–171
subroutines
  in C, see C functions
  in FORTRAN, see FORTRAN subroutines
  in IDL, see IDL functions, IDL procedures
  in MAPLE, see MAPLE functions
suppressing evaluation
  in MAPLE, see inert commands, 116
suppressing output
  in IDL, 20
  in MAPLE, 92

T
tables
  in \LaTeX, 472–474
  in MAPLE, 94
Taylor series
  for solving ODEs, 270–274
  for solving simultaneous equations, 442
  in MAPLE, 115, 144, 147, 153
termination of statements
  in C, 181
  in IDL, 16
  in MAPLE, 92
  in pseudocode, 162
\TeX, 457, 465
\TeX contact information, 457, 509
\TeX output
  from MAPLE, see \LaTeX output
texdoc, 477, 499
\TeXlive, 1
\TeXlive contact information, 509
text editors, see Local Guide
text in graphics, see placing text in graphics
TGIF, 1
TGIF contact information, 509
thermal equilibrium, 85
thin lens equation, 83
trajectory, 329
tikz, 477–480
tikz commands
  \definecolor, 478
  \draw, 478–479
  \foreach, 478, 479
  \node, 478, 479
  \x, 478, 479
trajectories, 329
  double pendulum, 315
earth falling into sun, 383, 387
exponentially decaying force, 383
in central forces, 251–253, 287–290, 319
in constant magnetic field, 69, 140, 212
in Coulomb field, 316
in crossed E and B, 319
simple harmonic oscillator, 383
under constant force, 338, 383
transformation rules
in MAPLE, 97
transmission at dielectric film, 83
transpose
in IDL, 22, 24–26
in MAPLE, 112, 127
trapezoidal rule, see quadrature
tree-structured directories, 8
tridiagonal systems, 439, 450
truncation error
global, 274
in midpoint formula, 354
in numerical quadrature, 353, 375, 381
in ODEs, 273, 277–278
in Simpson’s rule, 354
in trapezoidal rule, 354, 359, 373, 379
per step, 274
truth table, 161
two’s complement notation, 6
two-way selection
in C, 185, 432
in FORTRAN, 178, 426
in IDL, 32–33
in MAPLE, 126
in pseudocode, 162

U

UNIX, 1
UNIX commands
aspell, 489–490
cc, 184, 240, 241, 308, 311, 313, 431, 433, 434, 436
cleanbak, 498
cleanTEX, 498
cp, 239, 240
dvipdfm, 462
dvips, 461
f77, 177, 239, 301, 305, 306, 373, 375, 425, 427, 429
f90, 177
f95, 177
ispell, 489–490
latex, 460, 461
lp, 462
ls, 485

pdfcrop, 484–485
pdflatex, 461
pdf2ps, 485
ps2pdf, 462
texdoc, 477
xdvi, 462
UNIX shell scripts
ceps2pdf, 484–485, 504
cleanbak, 498
cleanTEX, 498
cps2pdf, 484–485, 504
rdfileeps, 485, 504
rdfileps, 485, 504
UNIX special symbols
> (output redirection), 177, 184
UNIX tutorial, see Local Guide
unstable equilibrium, 396
user accounts, see Local Guide
user-defined commands, see command files
user-defined functions
in C, see C functions
in FORTRAN, see FORTRAN functions
in IDL, see IDL functions
in MAPLE, see MAPLE functions
user-defined procedures
in IDL, see IDL procedures
user-defined subroutines
in C, see C functions
in FORTRAN, see FORTRAN subroutines

V

Van der Pol oscillator, 319
variable names
in C, 181
in FORTRAN, 173, 175
in IDL, 18
in MAPLE, 95
in pseudocode, 158
vector arrays
three-dimensional, 202–204
two-dimensional, 196–202
vector calculus
with MAPLE, 122–123
vector fields
three-dimensional
in C, 203–204
in FORTRAN, 202–203
in IDL, 204
two-dimensional
in C, 198–199
in FORTRAN, 198
in IDL, 200–202
vector identities, 155
vectors
in MAPLE, 99–102
velocity field
of rotating object, 213
vim, 1
viscous damping, 245, 248

**W**

wave function, 254–255, 293–297, 336, 385, 386  
Wheatstone bridge, 148  
Windows, 1  
Windows batch files  
    ceps2pdf.bat, 484–485, 503  
    cleanbak.bat, 498  
    cleanTEX.bat, 498  
    cps2pdf.bat, 484–485, 503  
    rdfileeps.bat, 485, 503  
    rdfileps.bat, 485, 503  
Windows commands  
    cleanbak, 498  
    cleanTEX, 498  
    dir, 485  
    dvipdfm, 462  
    dvips, 461  
    latex, 460, 461  
    pdfcrop, 484–485  
    pdflatex, 461  
    pdftops, 485  
    print, 462  
    ps2pdf, 462  
    texdoc, 477  
  yap, 462  
Winedt, 1  
Winedt contact information, 509  
writing data to files, 212, 213  
    in C, 184, 190–191, 195, 198–199, 203–204  
    in FORTRAN, 177, 189, 194, 198, 202–203  
    in pseudocode, 170–173  
writing data to screen  
    in C, 182, 183  
    in FORTRAN, 174, 176  
    in IDL, 20–22  
    in pseudocode, 160  

**X**

xdvi, 462, 488–489  
Xemacs, 1  
Xemacs contact information, 510  
xmaple, 92  
Xv contact information, 510

**Y**

yap, 462, 488–489