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\(^1\) 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.\(^2\)

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.

\(^1\)The 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—be 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 discuss³ 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.

³Many 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. Gehrke, LU '93, Karl J. Geissler, LU '94, Steven Van Metre, LU '94, Alain Bellon, LU '95, Peter Kelly Senecal, LU '95, Christopher C. Schmidt, LU '97, Michael D. Stenner, LU '97, Mark Nornberg, LU '98, Scot Shaw, LU '98, Jim Truitt, LU '98, Eric D. Moore, LU '99, Teresa K. Hayne, LU '00, Danica Dralus, LU, '02, Ryan T. Peterson, LU '03, Scott J. Kaminski, LU '04, Michelle L. Milne, LU '04, Lauren E. Kost, LU '05, Claire Weiss, LU '07, and Erik Garback, 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\(^1\) 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, 

\(^1\)In July 2001: Steve Adams (Widener University), Russel Kauffman (Muhlenberg College), Marylin Bell (Lake Forest College), Lawrence A. Molnar (Calvin College), Thomas R. Greenlee (Bethel College), Elliott Moore (New Mexico Tech), Javier Hasbun (State University of West Georgia), Joelle L. Murray (Linfield College), Derrick Hylton (Spelman College), Paula C. Turner (Kenyon College), Ross Hyman (DePaul University), Scott N. Walck (Lebanon Valley College), William H. Ingham (James Madison University), and Tim Young (University of North Dakota). In July, 2002 (first workshop): Albert Batten (United States Air Force Academy), Andrea Cox LU '91 (Beloit College), Brian Cudnik (Prairie View A and M), Craig Gunsul (Whitman College), Kevin Lee (University of Nebraska), Kam-Biu Luk (University of California, Berkeley), Daryl Macomb (Boise State University), Viktor Martirosyants (Central College), Donald Miller (Central Missouri State University), Dorn Peterson (James Madison University), Robert Ragan (University of Wisconsin – Lacrosse), Shafiq Rahman (Allegheny College), Rahmathullah 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), Russell Kauffman (Muhlenberg College), Joelle L. Murray (Linfield College), Shafiq Rahman (Allegheny College), John Thompson (DePaul University), and Tim Young (University of North Dakota), all of whom used at least portions of the growing manuscript in their teaching during the academic year 2001–02 and who collectively sent me a wide spectrum of varied and valuable comments.

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

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

- Donald Knuth, originator of \TeX;
- Leslie Lamport, originator of \LaTeX;
- Han The Than, originator of pdf\TeX, which provides the engine underlying pdflatex for creating PDF files directly from \LaTeX source code;
- Pehong Chen and Nelson Beebe, originator and current maintainer of makeindex for preparing indices to \LaTeX documents;
- Enrico Gregorio, originator and current maintainer of imakeidx that reduces the number of passes required to format and include an index;
- David Carlisle and Sebastian Rahtz, originators, and David Carlisle, current maintainer, of the \LaTeX package 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 \LaTeX package hyperref for creating linked versions of documents as PDF files;
- Paul Vojta, author of xdvı, 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 workshop): Julio Blanco (California State University–Northridge), William Briscoe (George Washington University), Paul Bunson (Lawrence University), Albert Chen (Oklahoma Baptist University), K. Kelvin Cheng (Texas Tech University), Bob Davis (Taylor University), Mirela Fetea (University of Richmond), David Grumbine (St. Vincent College), Kristi Hendrickson LU ’91 (University of Puget Sound), Tom Herrmann (Eastern Oregon University), Seamus Lagan (Whittier College), Eric Lane (University of Tennessee at Chattanooga), Ernest Ma (Montclair State University), Norris Preyer (College of Charleston), Michael Schillaci (Francis Marion University), Mihir Sejpal (Earlham College), Laura Van Wormer (Hiram College), Mary Lou West (Montclair State University), and Douglas Young (Mercer University). In July, 2003: Stephen Addison (University of Central Arkansas), Mohan Aggarwal (Alabama A&M), Anne Caraley (SUNY at Oswego), William Dieterle (California University of Pennsylvania), Timothy Duman (University of Indianapolis), Brett Fadem (Colby College), Michael Gray (American University), James Meyer (St. Gregory’s University), Prabasaj Paul (Denison University), David Peterson (Francis Marion University), Edward Pogozelski (SUNY at Geneseo), Chandra Prayaga (University of West Florida), Bruce Richards (Oberlin College), David Seely (Albion College), Adam Smith (Hillsdale College), Walther Spjeldvik (Weber State University), Julie Talbot (State University of West Georgia), Paul Thomas (University of Wisconsin–Eau Claire), Srinivasa Venugopalan (SUNY at Binghamton), and Alma Zook (Pomona College).
ACKNOWLEDGEMENTS

• The many contributors to the MiKTeX project;
• The developers, authors, and maintainers of WinEdit, 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 specifically flmoon, xflmoon, caldat, julday, xjulday, avervar, xavevar, rk4, xrk4, rkqs, xrkq, mmid, bestep, rkdumb, odeint, trapzd, xtrapzd, qtrap, xqtrap, qsimp, qcomb, polint, rbis, xrbis, rtnewt, xrtnewt, rtsafe, xrtsafe, zbrak, gauss, ludcmp, lubksb, tridag, svdcmp, svbksb, mmnewt, newt, and breydn (both in FORTRAN and in C).
3 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.\footnote{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 Mathematica details herein apply specifically to Mathematica Version 11 on a Hewlett-Packard platform running Windows 7 and a Hewlett-Packard platform running the Fedora 17 implementation of LINUX.

- The PYTHON details herein have been tested with
  - PYTHON 2.7.16 from the Anaconda2 distribution installed on a Hewlett-Packard platform running Windows 10, using the Anaconda2 prompt and also using the Anaconda2 Python Shell.\footnote{2}
  - PYTHON 3.7.3 from the Anaconda3 distribution installed on a Hewlett-Packard platform running Windows 10, using the Anaconda3 prompt and also using Anaconda3 Python Shell.\footnote{3}

The PYTHON 2 and PYTHON 3 details herein have also been spot-checked using Spyder,
though the prompt for commands in Spyder differs from the prompt used in this text.

\footnote{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.}

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

\footnote{3}{See the previous footnote.}
• 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 \TeX details in Appendix A apply specifically to \TeX with the MiK\TeX implementation on a Hewlett-Packard platform running Windows 7. \TeX normally responds to the same source code on all platforms.
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

5 Introduction to PYTHON ............. 15
   5.1 Beginning a PYTHON Session ....... 16
   5.2 Basic Entities in PYTHON ...... 17
      5.2.1 Data Types ................ 17
      5.2.2 Variable Names ............. 17
      5.2.3 Assignment of Values to Variables ... 18
      5.2.4 Commands ................ 18
      5.2.5 PYTHON Modules .......... 19
   5.3 A Sampling of PYTHON Capabilities .... 20
      5.3.1 Using PYTHON Interactively .... 20
      5.3.2 Creating and Examining Strings, Tuples, Lists , and Sets . 22
      5.3.3 Interrogating and Adjusting the Symbol Table .... 24
      5.3.4 Creating and Examining Arrays .... 24
      5.3.5 Creating and Manipulating Matrices .... 27
      5.3.6 Complex Numbers ............. 28
      5.3.7 Solving Linear Equations ....... 29
      5.3.8 Data and Method Attributes of Variables ... 30
      5.3.9 A First Graph .............. 30
   5.4 Loops, Logical Expressions, and Conditionals .... 33
   5.5 Eigenvalues and Eigenvectors .... 36
   5.6 The PYTHON Edit Window and Stored Programs .... 38
      5.6.1 A Quick Example of a Stored Program .... 38
      5.6.2 Reading Data From a File; Radioactive Decay .... 40
      5.6.3 Writing Data To a File; Formatting Output .... 41
   5.7 Defining Functions .............. 43
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.7.1</td>
<td>Format of a Function Definition</td>
<td>43</td>
</tr>
<tr>
<td>5.7.2</td>
<td>The Function <code>luplot</code></td>
<td>44</td>
</tr>
<tr>
<td>5.7.3</td>
<td>Using Global Variables</td>
<td>45</td>
</tr>
<tr>
<td>5.8</td>
<td>On-Line Help</td>
<td>48</td>
</tr>
<tr>
<td>5.9</td>
<td>Graphing Scalar Functions of One Variable</td>
<td>48</td>
</tr>
<tr>
<td>5.9.1</td>
<td>The Basic Strategy</td>
<td>49</td>
</tr>
<tr>
<td>5.9.2</td>
<td>Plotting Several Graphs on One Set of Axes</td>
<td>50</td>
</tr>
<tr>
<td>5.9.3</td>
<td>Polar Plots and Custom Axes</td>
<td>52</td>
</tr>
<tr>
<td>5.9.4</td>
<td>Multiple Plots In a Single Window</td>
<td>53</td>
</tr>
<tr>
<td>5.9.5</td>
<td>Plotting Experimental Data</td>
<td>54</td>
</tr>
<tr>
<td>5.10</td>
<td>Making Hard Copy</td>
<td>57</td>
</tr>
<tr>
<td>5.10.1</td>
<td>. . . of Text</td>
<td>57</td>
</tr>
<tr>
<td>5.10.2</td>
<td>. . . of Graphs</td>
<td>57</td>
</tr>
<tr>
<td>5.11</td>
<td>Graphing Scalar Functions of Two Variables</td>
<td>58</td>
</tr>
<tr>
<td>5.11.1</td>
<td>A Preliminary: The Function <code>meshgrid</code></td>
<td>59</td>
</tr>
<tr>
<td>5.11.2</td>
<td>Surface Plots: The Commands <code>plot_wireframe</code> and <code>plot_surface</code></td>
<td>60</td>
</tr>
<tr>
<td>5.11.3</td>
<td>Contour Plots: The Command <code>contour</code></td>
<td>63</td>
</tr>
<tr>
<td>5.11.4</td>
<td>Shaded Surfaces: The Command <code>plot_surface</code> Again</td>
<td>65</td>
</tr>
<tr>
<td>5.11.5</td>
<td>Features of the Figure Window</td>
<td>67</td>
</tr>
<tr>
<td>5.11.6</td>
<td>Functions of Two Variables in Polar Coordinates</td>
<td>67</td>
</tr>
<tr>
<td>5.12</td>
<td>Graphing Scalar Functions of Three Variables</td>
<td>68</td>
</tr>
<tr>
<td>5.12.1</td>
<td>Reduction to Two-Dimensional Displays</td>
<td>70</td>
</tr>
<tr>
<td>5.12.2</td>
<td>Isosurfaces</td>
<td>72</td>
</tr>
<tr>
<td>5.13</td>
<td>Graphing Vector Fields</td>
<td>73</td>
</tr>
<tr>
<td>5.13.1</td>
<td>The Function <code>quiver</code></td>
<td>75</td>
</tr>
<tr>
<td>5.13.2</td>
<td>More Elaborate Displays</td>
<td>76</td>
</tr>
<tr>
<td>5.13.3</td>
<td>Three-Dimensional Vector Fields</td>
<td>77</td>
</tr>
<tr>
<td>5.14</td>
<td>Animation</td>
<td>78</td>
</tr>
<tr>
<td>5.14.1</td>
<td>The Basic Command</td>
<td>78</td>
</tr>
<tr>
<td>5.14.2</td>
<td>Motion of a Vibrating String</td>
<td>79</td>
</tr>
<tr>
<td>5.14.3</td>
<td>Motion of an L-Shaped Drumhead</td>
<td>80</td>
</tr>
<tr>
<td>5.14.4</td>
<td>Trajectory of a Projectile in Uniform Gravity</td>
<td>82</td>
</tr>
<tr>
<td>5.15</td>
<td>Advanced Graphing Features</td>
<td>84</td>
</tr>
<tr>
<td>5.15.1</td>
<td>Mathematical Symbols and Greek Characters</td>
<td>84</td>
</tr>
<tr>
<td>5.15.2</td>
<td>Space Curves</td>
<td>85</td>
</tr>
<tr>
<td>5.15.3</td>
<td>Using Multiple Windows</td>
<td>85</td>
</tr>
<tr>
<td>5.15.4</td>
<td>Scaling so Circles are Circles</td>
<td>86</td>
</tr>
<tr>
<td>5.15.5</td>
<td>Customizing Axes</td>
<td>86</td>
</tr>
<tr>
<td>5.15.6</td>
<td>Alternative Specifications of Color</td>
<td>88</td>
</tr>
<tr>
<td>5.15.7</td>
<td>Interactive Plotting</td>
<td>90</td>
</tr>
<tr>
<td>5.16</td>
<td>Miscellaneous Occasionally Useful Tidbits</td>
<td>90</td>
</tr>
<tr>
<td>5.16.1</td>
<td>Find, Change Current Directory</td>
<td>90</td>
</tr>
<tr>
<td>5.16.2</td>
<td>Setting the Search Path</td>
<td>91</td>
</tr>
<tr>
<td>5.16.3</td>
<td>Customizing PYTHON</td>
<td>92</td>
</tr>
<tr>
<td>5.16.4</td>
<td>Restoring PYTHON's Initial State</td>
<td>94</td>
</tr>
<tr>
<td>5.16.5</td>
<td>Reading Data from the Keyboard</td>
<td>94</td>
</tr>
<tr>
<td>5.16.6</td>
<td>Error Trapping</td>
<td>95</td>
</tr>
<tr>
<td>5.17</td>
<td>References</td>
<td>95</td>
</tr>
<tr>
<td>5.18</td>
<td>Exercises</td>
<td>96</td>
</tr>
<tr>
<td>5.18.1</td>
<td>Writing PYTHON Statements</td>
<td>96</td>
</tr>
<tr>
<td>5.18.2</td>
<td>Finding Eigenvalues and Eigenvectors</td>
<td>97</td>
</tr>
<tr>
<td>5.18.3</td>
<td>Graphing Scalar Functions of a Single Variable</td>
<td>98</td>
</tr>
<tr>
<td>5.18.4</td>
<td>Graphing Scalar Functions of Two Variables</td>
<td>100</td>
</tr>
<tr>
<td>5.18.5</td>
<td>Graphing Scalar Functions of Three Variables</td>
<td>102</td>
</tr>
<tr>
<td>5.18.6</td>
<td>Graphing Vector Fields</td>
<td>102</td>
</tr>
<tr>
<td>5.18.7</td>
<td>Animation</td>
<td>104</td>
</tr>
<tr>
<td>5.A</td>
<td>Listing of <code>plotradio.py</code></td>
<td>106</td>
</tr>
<tr>
<td>5.B</td>
<td>Listing of <code>rcfilter.py</code></td>
<td>106</td>
</tr>
<tr>
<td>5.C</td>
<td>Listing of <code>stwave.py</code></td>
<td>107</td>
</tr>
<tr>
<td>5.D</td>
<td>Listing of <code>drumhead.py</code></td>
<td>108</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS

8 Introduction to MATHEMATICA .............................................................. 111

8.1 Beginning a Mathematica Session ...................................................... 112
8.2 On-Line Help ......................................................................................... 113
8.3 Basic Entities in Mathematica ............................................................... 114
8.4 Variable Names in Mathematica ............................................................ 114
8.5 Expressions in Mathematica ................................................................. 115
8.6 Assigning Values to Variables; Defining Functions ............................. 116
8.7 Creating and Examining Lists ............................................................... 119
8.8 Using Mathematica ................................................................................. 120
  8.8.1 . . . for Arithmetic ........................................................................ 121
  8.8.2 . . . for Algebra ........................................................................... 122
  8.8.3 . . . for Complex Variables ........................................................... 125
  8.8.4 . . . for Trigonometry ................................................................. 127
  8.8.5 . . . for Algebraic Equations ........................................................ 128
  8.8.6 . . . for Manipulating Vectors and Matrices ................................. 130
  8.8.7 . . . for Basic Calculus ............................................................... 131
  8.8.8 . . . for Laplace Transforms ......................................................... 134
  8.8.9 . . . for Ordinary Differential Equations ..................................... 135
  8.8.10 . . . for Vector Calculus ............................................................. 139
8.9 Packages ............................................................................................... 141
8.10 Loops, Logical Expressions, and Conditionals .................................. 142
8.11 Command Files ................................................................................... 144
8.12 High-Resolution Graphing ................................................................. 146
8.13 Making Hard Copy .............................................................................. 149
  8.13.1 . . . from the NBI ....................................................................... 150
  8.13.2 . . . from the TBI ....................................................................... 152
8.14 Output in \LaTeX{} Format ................................................................. 152
8.15 Animation ........................................................................................... 153
8.16 Using the Notebook ............................................................................ 154
8.17 Customizations .................................................................................. 155
8.18 Miscellaneous Occasionally Useful Tidbits .................................... 156
  8.18.1 Turning off Messages .................................................................. 156
  8.18.2 Specifying Directories ............................................................... 156
  8.18.3 Default Directory ....................................................................... 156
  8.18.4 Search Path ................................................................................ 157
  8.18.5 Displaying and Changing Default Options ................................ 157
  8.18.6 Placing Text in Graphs .............................................................. 158
  8.18.7 Space Curves ............................................................................. 158
  8.18.8 Directing Plot to Separate Window ........................................... 159
8.19 References ......................................................................................... 160
8.20 Exercises ........................................................................................... 160

9 Introduction to Programming ............................................................... 171

9.1 Components of a Programming Language ....................................... 172
  9.1.1 Variables, Variable Names, and Use of Memory ....................... 172
  9.1.2 Essential Action Statements ....................................................... 173
  9.1.3 Logical Expressions and Conditions ....................................... 174
  9.1.4 Syntactic Wrinkles .................................................................... 175
  9.1.5 Essential Control Structures ..................................................... 176
9.2 Sample Short Algorithms .................................................................. 180
9.3 Two Larger Algorithms ..................................................................... 182
  9.3.1 Solving Laplace’s Equation ....................................................... 182
  9.3.2 File Output/Input ..................................................................... 184
9.4 Solving Laplace’s Equation with FORTRAN ................................... 187
9.5 Additional Features of FORTRAN .................................................... 191
  9.5.1 Multi-Way Selection .................................................................. 191
  9.5.2 Reading Data from Keyboard ................................................... 192
  9.5.3 Additional Loops ....................................................................... 193
# Table of Contents

## 9.5.4 Subroutines, Functions, and Common Storage ........................................ 193

## 9.6 Solving Laplace's Equation with C ......................................................... 194

## 9.7 Additional Features of C ............................................................................. 198

### 9.7.1 Multi-Way Selection ............................................................................. 198

### 9.7.2 Reading Data from Keyboard .............................................................. 199

### 9.7.3 Additional Loops .................................................................................. 200

### 9.7.4 Functions and Global Variables ......................................................... 200

## 9.11 Solving Laplace's Equation with PYTHON .................................................. 201

## 9.12 Creating and Storing Two-Dimensional Scalar Arrays .............................. 203

### 9.12.1 ... with FORTRAN ........................................................................... 203

### 9.12.2 ... with C ......................................................................................... 204

### 9.12.6 ... with PYTHON .............................................................................. 206

## 9.13 Creating and Storing Three-Dimensional Scalar Arrays ............................ 208

### 9.13.1 ... with FORTRAN ........................................................................... 209

### 9.13.2 ... with C ......................................................................................... 209

### 9.13.6 ... with PYTHON .............................................................................. 210

## 9.14 Creating and Storing Two-Dimensional Vector Arrays ............................... 211

### 9.14.1 ... with FORTRAN ........................................................................... 212

### 9.14.2 ... with C ......................................................................................... 213

### 9.14.6 ... with PYTHON .............................................................................. 214

## 9.15 Creating and Storing Three-Dimensional Vector Arrays ............................ 217

### 9.15.1 ... with FORTRAN ........................................................................... 218

### 9.15.2 ... with C ......................................................................................... 218

### 9.15.6 ... with PYTHON .............................................................................. 219

## 9.16 Reading Files ............................................................................................. 220

### 9.16.1 ... with FORTRAN ........................................................................... 220

### 9.16.2 ... with C ......................................................................................... 221

### 9.16.6 ... with PYTHON .............................................................................. 222

## 9.17 References .................................................................................................. 223

## 9.18 Exercises ..................................................................................................... 223

## 9.9 Listing of irrad.f ......................................................................................... 230

## 9.10 Listing of irrad.c ....................................................................................... 231

## 9.11 Listing of irrad.py ..................................................................................... 233

## 9.12 Listing of pdens.f ..................................................................................... 234

## 9.13 Listing of pdens.c ..................................................................................... 236

## 9.14 Listing of pdens.py ................................................................................... 238

## 9.15 Listing of wavegd.f .................................................................................. 240

## 9.16 Listing of wavegd.c .................................................................................. 242

## 9.17 Listing of wavegd.py ................................................................................ 244

## 9.18 Listing of quadpole.f ................................................................................. 245

## 9.19 Listing of quadpole.c ................................................................................ 247

## 9.20 Listing of quadpole.py ............................................................................... 249

## 9.21 Listing of read_irrad.f .............................................................................. 251

## 9.22 Listing of read_irrad.c .............................................................................. 252

## 9.23 Listing of read_irrad.py ............................................................................ 252

## 10 Introduction to Numerical Recipes ................................................................. 255

### 10.1 The Numerical Recipes Directory Tree .................................................... 255

### 10.2 Using FORTRAN Recipes ...................................................................... 256

### 10.3 Using C Recipes ..................................................................................... 258

### 10.4 Exercises .................................................................................................. 259

## 11 Solving Ordinary Differential Equations ......................................................... 261

### 11.1 Sample Problems ...................................................................................... 262

#### 11.1.1 Projectile in a Viscous Medium .......................................................... 262

#### 11.1.2 Radioactive Decay ............................................................................. 265

#### 11.1.3 Exponential and Logistic Growth ....................................................... 266

#### 11.1.4 Forced, Driven, Damped Harmonic Oscillation ................................. 266

#### 11.1.5 An LRC Resonant Circuit ................................................................. 268

#### 11.1.6 Coupled Oscillators .......................................................................... 268
11.17 Motion under Central Forces ........................................... 269
11.18 Standing Waves in a String ........................................... 271
11.19 The Schrödinger Equation in One Dimension ......................... 272
11.2 Laplace Transforms .................................................. 273
11.5 Solving ODEs Symbolically with Mathematica ......................... 274
  11.5.1 Projectile in a Viscous Medium .................................. 275
  11.5.2 Logistic Growth .................................................. 277
  11.5.3 Damped Harmonic Oscillator .................................... 278
  11.5.4 Chain Radioactive Decay ........................................ 278
  11.5.5 Coupled Oscillators ............................................. 281
  11.5.6 Standing Waves in a String ...................................... 282
  11.5.7 Infinite Depth Quantum Well .................................... 285
  11.5.8 Finding Solutions by Series Methods ............................ 285
11.6 Algorithms for Solving ODEs Numerically ............................ 288
  11.6.1 Euler’s Method .................................................. 288
  11.6.2 Improved Euler Method .......................................... 290
  11.6.3 Runge-Kutta Methods ............................................ 292
  11.6.4 Assessing Accuracy .............................................. 294
  11.6.5 Adaptive Methods .............................................. 295
  11.6.6 Multistep Methods .............................................. 296
11.10 Solving ODEs Numerically with PYTHON ............................ 297
  11.10.1 Using Elementary Commands .................................... 297
  11.10.2 The PYTHON Command odeint .................................... 299
  11.10.3 Defining ODEs for odeint ...................................... 301
  11.10.4 Chain Radioactive Decay ........................................ 302
  11.10.5 Damped Harmonic Oscillator .................................... 305
  11.10.6 Planetary Orbits .............................................. 306
  11.10.7 Standing Waves in a String ..................................... 310
  11.10.8 The Quantum Harmonic Oscillator ................................ 313
  11.10.9 Keywords that Modify odeint .................................... 318
  11.10.10 py-files in the Public Library ................................. 319
  11.10.11 Other ODE Solvers in PYTHON ................................. 320
11.13 Solving ODEs Numerically with Mathematica ........................ 320
11.14 Solving ODEs Numerically with FORTRAN .......................... 322
  11.14.1 FORTRAN Programs from Scratch .............................. 323
  11.14.2 Using Numerical Recipes ....................................... 324
11.15 Solving ODEs Numerically with C .................................. 329
  11.15.1 C Programs from Scratch ....................................... 329
  11.15.2 Using Numerical Recipes ....................................... 331
11.16 Exercises .................................................................. 336
  11.16.1 ... using Symbolic Methods ..................................... 336
  11.16.2 ... using Numerical Methods .................................... 340
  11.16.3 ... using Numerical Recipes ..................................... 343
11.A Listing of decay.f .................................................... 344
11.B Listing of decayrk4.f .................................................. 345
11.C Listing of decayrk4qs.f ............................................... 346
11.D Listing of decay.c ..................................................... 347
11.E Listing of decayrk4.c .................................................. 348
11.F Listing of decayrk4qs.c ............................................... 349
13 Evaluating Integrals .................................................................. 351
  13.1 Sample Problems ................................................................ 351
    13.1.1 One-Dimensional Trajectories ................................ 351
    13.1.2 Center of Mass .................................................... 352
    13.1.3 Moment of Inertia; Radius of Gyration ....................... 353
    13.1.4 The Large Amplitude Simple Pendulum ....................... 354
    13.1.5 Statistical Data Analysis ........................................ 355
    13.1.6 The Cornu Spiral .................................................. 356
    13.1.7 Electric and Magnetic Fields and Potentials .................. 356
    13.1.8 Quantum Probabilities ........................................... 358
<table>
<thead>
<tr>
<th>13.4 Evaluating Integrals Symbolically with Mathematica</th>
<th>419</th>
</tr>
</thead>
<tbody>
<tr>
<td>13.4.1 Relativistic Motion Under a Constant Force</td>
<td>360</td>
</tr>
<tr>
<td>13.4.2 Center of Mass</td>
<td>361</td>
</tr>
<tr>
<td>13.4.3 Moment of Inertia: Radius of Gyration</td>
<td>362</td>
</tr>
<tr>
<td>13.4.4 Electrostatic Potential of a Finite Line Charge</td>
<td>362</td>
</tr>
<tr>
<td>13.4.5 The Helmholtz Coil</td>
<td>364</td>
</tr>
<tr>
<td>13.4.6 Period of a Pendulum: Correction as Amplitude Grows</td>
<td>368</td>
</tr>
<tr>
<td>13.4.7 Fourier Coefficients for Half-Rectified Signal</td>
<td>369</td>
</tr>
<tr>
<td>13.5 Algorithms for Numerical Integration</td>
<td>372</td>
</tr>
<tr>
<td>13.5.1 Newton-Cotes Quadrature</td>
<td>372</td>
</tr>
<tr>
<td>13.5.2 Rearrangements for Computational Efficiency</td>
<td>373</td>
</tr>
<tr>
<td>13.5.3 Assessing Error</td>
<td>375</td>
</tr>
<tr>
<td>13.5.4 Iterative and Adaptive Algorithms</td>
<td>377</td>
</tr>
<tr>
<td>13.5.5 Gaussian Quadrature</td>
<td>377</td>
</tr>
<tr>
<td>13.9 Evaluating Integrals Numerically with PYTHON</td>
<td>380</td>
</tr>
<tr>
<td>13.9.1 Using Elementary Commands</td>
<td>380</td>
</tr>
<tr>
<td>13.9.2 Built-In Integration Routines</td>
<td>383</td>
</tr>
<tr>
<td>13.9.3 Moment of Inertia</td>
<td>386</td>
</tr>
<tr>
<td>13.9.4 Quantum Probabilities</td>
<td>387</td>
</tr>
<tr>
<td>13.9.5 Integrals as Functions of the Upper Limit</td>
<td>387</td>
</tr>
<tr>
<td>13.9.6 The Error Function</td>
<td>388</td>
</tr>
<tr>
<td>13.9.7 The Cornu Spiral</td>
<td>388</td>
</tr>
<tr>
<td>13.9.8 Integrals as Functions of an Internal Parameter</td>
<td>390</td>
</tr>
<tr>
<td>13.9.9 The Off-Axis Electrostatic Potential of Two Rings</td>
<td>391</td>
</tr>
<tr>
<td>13.9.10 Keywords that Modify Function Integrators</td>
<td>392</td>
</tr>
<tr>
<td>13.12 Evaluating Integrals Numerically with Mathematica</td>
<td>393</td>
</tr>
<tr>
<td>13.12.1 Quantum Probability</td>
<td>395</td>
</tr>
<tr>
<td>13.12.2 The Error Function</td>
<td>395</td>
</tr>
<tr>
<td>13.12.3 The Off-Axis Electrostatic Potential of Two Rings</td>
<td>395</td>
</tr>
<tr>
<td>13.13 Evaluating Integrals Numerically with FORTRAN</td>
<td>396</td>
</tr>
<tr>
<td>13.13.1 Writing Programs from Scratch</td>
<td>396</td>
</tr>
<tr>
<td>13.13.2 Using Numerical Recipes</td>
<td>399</td>
</tr>
<tr>
<td>13.14 Evaluating Integrals Numerically with C</td>
<td>402</td>
</tr>
<tr>
<td>13.14.1 Writing Programs from Scratch</td>
<td>402</td>
</tr>
<tr>
<td>13.14.2 Using Numerical Recipes</td>
<td>404</td>
</tr>
<tr>
<td>13.15 Exercises</td>
<td>407</td>
</tr>
<tr>
<td>13.15.1 ... using Symbolic Methods</td>
<td>407</td>
</tr>
<tr>
<td>13.15.2 ... using Numerical Methods</td>
<td>410</td>
</tr>
<tr>
<td>13.15.3 ... using Numerical Recipes</td>
<td>414</td>
</tr>
<tr>
<td>13.A Listing of trapezoidal.f</td>
<td>416</td>
</tr>
<tr>
<td>13.B Listing of trapezoidal.c</td>
<td>417</td>
</tr>
<tr>
<td>14 Finding Roots</td>
<td>419</td>
</tr>
<tr>
<td>14.1 Sample Problems</td>
<td>419</td>
</tr>
<tr>
<td>14.1.1 Classical Turning Points</td>
<td>420</td>
</tr>
<tr>
<td>14.1.2 A Max-Min Problem: Equilibrium Points</td>
<td>420</td>
</tr>
<tr>
<td>14.1.3 Natural Frequencies of Oscillating Systems</td>
<td>421</td>
</tr>
<tr>
<td>14.1.4 Range of Projectile in Viscous Medium</td>
<td>422</td>
</tr>
<tr>
<td>14.1.5 Energy Levels in a Quantum Well</td>
<td>425</td>
</tr>
<tr>
<td>14.2 Symbolic Approaches</td>
<td>426</td>
</tr>
<tr>
<td>14.5 Finding Roots Symbolically with Mathematica</td>
<td>427</td>
</tr>
<tr>
<td>14.5.1 Classical Turning Points</td>
<td>428</td>
</tr>
<tr>
<td>14.5.2 Equilibrium Points</td>
<td>428</td>
</tr>
<tr>
<td>14.5.3 Natural Frequencies</td>
<td>429</td>
</tr>
<tr>
<td>14.5.4 Range of Projectile in Viscous Medium</td>
<td>430</td>
</tr>
<tr>
<td>14.6 Algorithms for Finding Roots Numerically</td>
<td>432</td>
</tr>
<tr>
<td>14.6.1 The Method of Bisection</td>
<td>432</td>
</tr>
<tr>
<td>14.6.2 Newton’s Method</td>
<td>433</td>
</tr>
<tr>
<td>14.6.3 Other Methods</td>
<td>433</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>A.1 Creating a Simple Document</td>
<td>484</td>
</tr>
<tr>
<td>A.1.1 Structuring a \TeX{} Source File</td>
<td>484</td>
</tr>
<tr>
<td>A.1.2 Creating a PostScript File</td>
<td>486</td>
</tr>
<tr>
<td>A.1.3 Creating a PDF File</td>
<td>487</td>
</tr>
<tr>
<td>A.2 Displaying the Document on the Screen</td>
<td>488</td>
</tr>
<tr>
<td>A.1.4 Creating a PDF File</td>
<td>488</td>
</tr>
<tr>
<td>A.1.5 Printing the Document</td>
<td>488</td>
</tr>
<tr>
<td>A.3 Specifying Global Style - The Preamble</td>
<td>489</td>
</tr>
<tr>
<td>A.3.1 Type Style</td>
<td>491</td>
</tr>
<tr>
<td>A.3.2 Type Size</td>
<td>492</td>
</tr>
<tr>
<td>A.3.3 White Space</td>
<td>493</td>
</tr>
<tr>
<td>A.3.4 Environments</td>
<td>493</td>
</tr>
<tr>
<td>A.3.5 \LaTeX{} Packages</td>
<td>495</td>
</tr>
<tr>
<td>A.3.6 Miscellaneous Other Capabilities</td>
<td>495</td>
</tr>
<tr>
<td>A.4 Including Equations</td>
<td>496</td>
</tr>
<tr>
<td>A.5 Including Lists</td>
<td>497</td>
</tr>
<tr>
<td>A.6 Including Tables</td>
<td>498</td>
</tr>
<tr>
<td>A.7 Including Illustrations</td>
<td>500</td>
</tr>
<tr>
<td>A.7.1 Using Cut and Paste</td>
<td>500</td>
</tr>
<tr>
<td>A.7.2 Using the graphicx Package</td>
<td>501</td>
</tr>
<tr>
<td>A.7.3 Using the tikz Package</td>
<td>503</td>
</tr>
<tr>
<td>A.7.4 Using the picture Environment and pict2e Package</td>
<td>506</td>
</tr>
<tr>
<td>A.8 Including TOC, LOF, and LOT</td>
<td>506</td>
</tr>
<tr>
<td>A.9 Including an Index</td>
<td>507</td>
</tr>
<tr>
<td>A.10 Including Hyperlinks</td>
<td>509</td>
</tr>
<tr>
<td>A.11 Converting .eps and .ps Files to .pdf</td>
<td>510</td>
</tr>
<tr>
<td>A.12 Using Conditional Expressions in \LaTeX{}</td>
<td>512</td>
</tr>
<tr>
<td>A.13 Error Messages Generated by \LaTeX{}</td>
<td>513</td>
</tr>
<tr>
<td>A.14 The Page Previewer for .dvi Files</td>
<td>514</td>
</tr>
<tr>
<td>A.15 The Spell Checker in UNIX</td>
<td>515</td>
</tr>
<tr>
<td>A.16 A Sample Document</td>
<td>516</td>
</tr>
<tr>
<td>A.17 Miscellaneous Other Features</td>
<td>517</td>
</tr>
<tr>
<td>14.6.4 Assessing Accuracy</td>
<td>434</td>
</tr>
<tr>
<td>14.10 Finding Roots Numerically with PYTHON</td>
<td>435</td>
</tr>
<tr>
<td>14.10.1 Classical Turning Points</td>
<td>436</td>
</tr>
<tr>
<td>14.10.2 Natural Frequencies</td>
<td>438</td>
</tr>
<tr>
<td>14.10.3 Range of Projectile</td>
<td>439</td>
</tr>
<tr>
<td>14.10.4 Energy Levels in Quantum Well</td>
<td>442</td>
</tr>
<tr>
<td>14.13 Finding Roots Numerically with Mathematica</td>
<td>444</td>
</tr>
<tr>
<td>14.13.1 Classical Turning Points</td>
<td>446</td>
</tr>
<tr>
<td>14.13.2 Range of Projectile</td>
<td>447</td>
</tr>
<tr>
<td>14.13.3 Energy Levels in Quantum Well</td>
<td>448</td>
</tr>
<tr>
<td>14.14 Finding Roots Numerically with FORTRAN</td>
<td>450</td>
</tr>
<tr>
<td>14.14.1 FORTRAN Programs from Scratch</td>
<td>450</td>
</tr>
<tr>
<td>14.14.2 Using Numerical Recipes</td>
<td>453</td>
</tr>
<tr>
<td>14.15 Finding Roots Numerically with C</td>
<td>457</td>
</tr>
<tr>
<td>14.15.1 C Programs from Scratch</td>
<td>457</td>
</tr>
<tr>
<td>14.15.2 Using Numerical Recipes</td>
<td>460</td>
</tr>
<tr>
<td>14.16 Solving Simultaneous Equations</td>
<td>463</td>
</tr>
<tr>
<td>14.16.1 Systems of Linear Equations</td>
<td>464</td>
</tr>
<tr>
<td>14.16.2 Systems of Nonlinear Equations</td>
<td>467</td>
</tr>
<tr>
<td>14.17 Exercises</td>
<td>470</td>
</tr>
<tr>
<td>14.17.1 ... using Symbolic Methods</td>
<td>470</td>
</tr>
<tr>
<td>14.17.2 ... using Numerical Methods</td>
<td>471</td>
</tr>
<tr>
<td>14.17.3 ... using Numerical Recipes</td>
<td>475</td>
</tr>
<tr>
<td>14.17.4 Finding More than One Unknown</td>
<td>475</td>
</tr>
<tr>
<td>14.A Listing of bisect.f</td>
<td>480</td>
</tr>
<tr>
<td>14.B Listing of bisect.c</td>
<td>481</td>
</tr>
<tr>
<td>14.17.4 Finding More than One Unknown</td>
<td>475</td>
</tr>
<tr>
<td>14.16.2 Systems of Nonlinear Equations</td>
<td>467</td>
</tr>
<tr>
<td>14.17.2 ... using Numerical Methods</td>
<td>471</td>
</tr>
<tr>
<td>14.17.3 ... using Numerical Recipes</td>
<td>475</td>
</tr>
<tr>
<td>14.17.4 Finding More than One Unknown</td>
<td>475</td>
</tr>
<tr>
<td>14.A Listing of bisect.f</td>
<td>480</td>
</tr>
<tr>
<td>14.B Listing of bisect.c</td>
<td>481</td>
</tr>
<tr>
<td>14.16 Solving Simultaneous Equations</td>
<td>463</td>
</tr>
<tr>
<td>14.16.1 Systems of Linear Equations</td>
<td>464</td>
</tr>
<tr>
<td>14.16.2 Systems of Nonlinear Equations</td>
<td>467</td>
</tr>
<tr>
<td>14.17 Exercises</td>
<td>470</td>
</tr>
<tr>
<td>14.17.1 ... using Symbolic Methods</td>
<td>470</td>
</tr>
<tr>
<td>14.17.2 ... using Numerical Methods</td>
<td>471</td>
</tr>
<tr>
<td>14.17.3 ... using Numerical Recipes</td>
<td>475</td>
</tr>
<tr>
<td>14.17.4 Finding More than One Unknown</td>
<td>475</td>
</tr>
<tr>
<td>14.A Listing of bisect.f</td>
<td>480</td>
</tr>
<tr>
<td>14.B Listing of bisect.c</td>
<td>481</td>
</tr>
<tr>
<td>14.15 Finding Roots Numerically with C</td>
<td>457</td>
</tr>
<tr>
<td>14.15.1 C Programs from Scratch</td>
<td>457</td>
</tr>
<tr>
<td>14.15.2 Using Numerical Recipes</td>
<td>460</td>
</tr>
<tr>
<td>14.16 Solving Simultaneous Equations</td>
<td>463</td>
</tr>
<tr>
<td>14.16.1 Systems of Linear Equations</td>
<td>464</td>
</tr>
<tr>
<td>14.16.2 Systems of Nonlinear Equations</td>
<td>467</td>
</tr>
<tr>
<td>14.17 Exercises</td>
<td>470</td>
</tr>
<tr>
<td>14.17.1 ... using Symbolic Methods</td>
<td>470</td>
</tr>
<tr>
<td>14.17.2 ... using Numerical Methods</td>
<td>471</td>
</tr>
<tr>
<td>14.17.3 ... using Numerical Recipes</td>
<td>475</td>
</tr>
<tr>
<td>14.17.4 Finding More than One Unknown</td>
<td>475</td>
</tr>
<tr>
<td>14.A Listing of bisect.f</td>
<td>480</td>
</tr>
<tr>
<td>14.B Listing of bisect.c</td>
<td>481</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.18 References</td>
<td>525</td>
</tr>
<tr>
<td>A.19 Exercises</td>
<td>525</td>
</tr>
<tr>
<td>A.A Listings</td>
<td>529</td>
</tr>
<tr>
<td>A.A.1 ... for Windows</td>
<td>529</td>
</tr>
<tr>
<td>A.A.2 ... for UNIX</td>
<td>530</td>
</tr>
<tr>
<td>A.A.3 ... for Windows and UNIX</td>
<td>531</td>
</tr>
<tr>
<td>Z Contacting Software Vendors</td>
<td>533</td>
</tr>
<tr>
<td>Index</td>
<td>537</td>
</tr>
</tbody>
</table>
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., T\TeXlive, E\TeX, MiK\TeX, Oz\TeX, ...) 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,
be motivated to use computational resources intelligently and successfully on their own initiative, whenever it seems to them appropriate to exploit those tools. As a resource, the computer should parallel the library; this book aims to help students develop the skills to support that view.\footnote{Uses of internet resources are conspicuously absent from the list of skills in this opening paragraph. While such uses are playing an increasingly important role both in education and in professional life, they are explicitly excluded from the purview of this book.}

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

1. You learn to manipulate the system you have and to work efficiently with whatever text editor is available. For the most part, this step is the task of the 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.
1.1.1 A Simple Responsive Machine

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

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

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

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

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

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

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

4 The special words IF, THEN, and END_IF convey a conditional execution of one or more statements. The statement(s) between the THEN and the END_IF will be executed only if the condition following the IF is true when the entire construction is encountered.
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,

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

5The number eight is, of course, arbitrary but conventional. Because computers work internally in the binary (base-2) number system, powers of two—8 = 2³—are especially convenient.

6Actually, the codes will be sent as a stream of bits, each of which is an electrical voltage level that will be either “high” or “low”, often said to be “on” or “off”.
1.1. AN ORIENTATION TO COMPUTERS

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

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

• a sixteen-bit combination of two consecutive bytes as an unsigned sixteen-bit integer, with its $2^{16} = 65536$ values assigned to the (positive) integers ranging (in decimal) from 0 to 65536.

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

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

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

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

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

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

---

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

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

<table>
<thead>
<tr>
<th>Code</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NUL</td>
</tr>
<tr>
<td>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>
<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>

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

In the IEEE standard for this format, eleven bits are assigned to store the exponent (as an eleven-bit signed integer, one bit is assigned to store the sign (0=+, 1=−) of the value, and 52 bits are assigned to store the digits of the (absolute value of the) value itself. In this format, values ranging (in decimal) from $2.225 \times 10^{-308}$ to $1.798 \times 10^{308}$ can be represented, though only to a precision of about fifteen decimal digits.\(^9\)

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

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

Any computer system will, of course, store a very large number of individual files. To keep these files under some semblance of control, they will commonly be grouped together into aggregates of various sizes, those aggregates will themselves be assembled into higher-level aggregates, those into still higher-level aggregates, and so on. 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, and so on. In the computer world, we need not then 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.

---

10The 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.

11Since 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.

12In 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.

13If 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.

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 *WinEdt* 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

```plaintext
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

```plaintext
R = 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

```plaintext
SQRTPRO( X^2 + Y^2, R )
```

having no variable or equal sign at its beginning.  

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* or *xf*, in the statement to be executed but render these variables as subscripted, e.g., *x*₁ or *x*₂, in the displayed output.

---

[14] In some languages, procedures can optionally be written *err = SQRTPRO(X^2+Y^2, R)*, in which case the procedure returns a value to the variable *err* to convey that the procedure encountered a problem in its execution. Testing *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 $\text{HEAD/idl}$, those associated with the chapter on integration will be found in subdirectories of the directory $\text{HEAD/integ}$, etc.

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

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

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

With regard to the index, please note the following:

- Italic page numbers in the index convey that the page reference is to an end-of-chapter exercise, not to a textual discussion.
- Because index commands in \TeX 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 .*, :, ", ==, /*...*/, and ! are included in the index. Since many of these symbols are, however, extremely common, the index includes pointers to only some of their appearances within the text. Those included have been chosen to illustrate a representative spectrum of uses, so reviewing the few items identified in the index should convey the full range of uses of these symbols.

1.5 Assumed Background

Mathematically, we assume in this book that you understand the notions of derivatives, integrals, and ordinary differential equations and that you have some acquaintance with linear algebra (matrix operations, eigenvalues and eigenvectors, ...). 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

\[
n! = n \times (n - 1) \times (n - 2) \times (n - 3) \times \cdots \times 2 \times 1
\]  
(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

\[
n!! = n \times (n - 2) \times (n - 4) \times (n - 6) \times \cdots \times \left\{ \begin{array}{ll} 4 \times 2 & n \text{ even} \\ 3 \times 1 & n \text{ odd} \end{array} \right\}
\]  
(1.2)

is less familiar but occurs frequently as well. Whether \( n \) is even or odd, the double factorial can be recast in terms of single factorials. If, for example, \( n \) is even, say 10, we can recast its double factorial in the form

\[
10!! = 10 \times 8 \times 6 \times 4 \times 2 = 2^5 \times 5 \times 4 \times 3 \times 2 \times 1 = 2^5 5!
\]  
\( \Rightarrow \)  
(2n)!! = \( 2^n \) n!

(1.3)

If \( n \) is odd, say 9, recasting its double factorial takes mildly more work, but is illustrated in the chain

\[
9!! = 9 \times 7 \times 5 \times 3 \times 1 = \frac{9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1}{8 \times 6 \times 4 \times 2} = \frac{9!}{8!!} = \frac{9!}{2^4 4!}
\]  
\( \Rightarrow \)  
(2n + 1)!! = \( \frac{(2n + 1)!}{2^n n!} \)

(1.4)

We might wonder whether it is possible to define a function of a continuous variable that will coincide with the factorial function when its argument is an integer. Consider the function \( \Gamma(\nu) \) defined by the integral

\[
\Gamma(\nu) = \int_0^\infty t^{\nu - 1} e^{-t} \, dt
\]  
(1.5)

We quickly conclude that

\[
\Gamma(1) = \int_0^\infty e^{-t} \, dt = 1
\]  
(1.6)

With a little more effort, we can evaluate \( \Gamma(1/2) \). We begin by writing the definition of \( \Gamma(1/2) \) and introducing the new variable \( x^2 = t \) in the definition, finding that

\[
\Gamma\left(\frac{1}{2}\right) = \int_0^\infty e^{-t} \, dt = \int_0^\infty e^{-x^2} \frac{x \, dx}{x} = \int_0^\infty e^{-x^2} \, dx
\]  
(1.7)

Then, we examine the square of the quantity of interest, change to polar coordinates, and find that

\[
\left[ \Gamma\left(\frac{1}{2}\right) \right]^2 = \left( \int_0^\infty e^{-x^2} \, dx \right)^2 = \left( \int_0^\infty e^{-x^2} \, dx \right) \left( \int_0^\infty e^{-y^2} \, dy \right)
\]  
(1.8)

\[
= \int_0^\infty \int_0^\infty e^{-(x^2+y^2)} \, dx \, dy = \int_0^\infty e^{-r^2} \, r \, dr \int_0^{2\pi} \, d\phi = \pi
\]  
(1.9)
and we conclude that
\[
\Gamma \left( \frac{1}{2} \right) = \sqrt{\pi} \tag{1.10}
\]

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
\[
\Gamma(\nu) = -\int_0^\infty t^{\nu-1} e^{-t} \, dt = \frac{1}{\nu} \int_0^\infty t^{\nu-2} e^{-t} \, dt = (\nu - 1) \Gamma(\nu - 1) \tag{1.11}
\]

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

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

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
\[
\mathcal{L} \left( f(t) \right) = \tilde{f}(s) = \int_0^\infty e^{-st} f(t) \, dt \tag{1.14}
\]
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,
\[
\mathcal{L} \left( af(t) + bg(t) \right) = a \int_0^\infty e^{-st} f(t) \, dt + b \int_0^\infty e^{-st} g(t) \, dt
\]
\[
= a \mathcal{L} (f(t)) + b \mathcal{L} (g(t)) \tag{1.15}
\]

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) - sx(0) - \frac{dx}{dt}(0) )</td>
</tr>
</tbody>
</table>

- The Laplace transform of the first derivative of a function \( f(t) \) is simply related to the Laplace transform of \( f(t) \). We need merely integrate the formal expression for the transform of the derivative by parts to find that

\[
\frac{df}{dt}(s) = \int_0^\infty e^{-st} \frac{df(t)}{dt} dt = e^{-s0} f(0) \bigg|_0^\infty + s \int_0^\infty e^{-st} f(t) dt = s \tilde{f}(s) - f(0) \tag{1.16}
\]

- The Laplace transform of a higher-order derivative is also simply related to the Laplace transform of the original function. We merely apply the identity in Eq. (1.16) repeatedly. The Laplace transform of a second derivative, for example, has the evaluation

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

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 5

Introduction to PYTHON

Note: All program (*.py) and data (*.dat) files referred to in this chapter are available in the directory \$HEAD/python, 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 PYTHON’s default search path. (See Section 5.16.2.) If so, the files can be found by PYTHON 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.

PYTHON is a freely distributed software package\(^1\) with capabilities to function as a very fancy interactive calculator, to run properly written programs, to generate graphical displays and animations, and to be essentially infinitely extended through the use of available or user-written modules.\(^2\) A typical interaction with the program will involve (1) creating one or more entities in PYTHON’s workspace (either by computing them directly in PYTHON or by importing them into PYTHON from a file created by another program), (2) processing them in some way to produce other entities, 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 these entities. Further details can be found in the on-line help messages accessible from within PYTHON itself (see Section 5.8) and in assorted PYTHON documentation (see Section 5.17). Throughout this chapter, we have scattered an occasional URL link to a specific topic in a footnote when the topic is introduced. We shall refer to all of this documentation collectively as the PYTHON manuals.

Currently, PYTHON is available in two versions. PYTHON 2.7, which is the most recent release of Version 2, was introduced in 2010. Because many programs existing for this version will require possibly significant recasting to work with the newer Version 3.x, the developers have agreed to continue supporting PYTHON 2.7 until 1 January 2020, although this version will receive no new major releases. PYTHON 3 is the forward-looking version, and PYTHON 3.6, which was released in 2016, is proclaimed by the developers to be stable. Many add-on modules written originally for PYTHON 2 have been ported to PYTHON 3, but many others have not yet been so ported. This chapter and examples in subsequent chapters have been written assuming that you are using PYTHON 2.7. Fortunately, much of what is here described will work also in PYTHON 3.6, and we will include warnings at any point where that is not the case.\(^3\)

\(^1\)See [www.python.org/psf/license](http://www.python.org/psf/license) for details on the license to which you must agree before downloading and installing PYTHON. Though you probably will not need access directly to the files in that distribution, they will on your system be installed in a directories whose top level directory we will refer to as \$PYTHONHEAD, whose explicit identification will be in your Local Guide.

\(^2\)See Appendix Z for full contact information.

\(^3\)A detailed comparison of PYTHON 2.7 with PYTHON 3.6 is provided on the web at the link [wiki.python.org/moin/Python2orPython3](http://wiki.python.org/moin/Python2orPython3). Review this document to help you decide which version will be best for your applications.
Two user interfaces to PYTHON are provided, a command-line interface (CLI) and a graphical user interface (GUI), called the Python shell, which facilitates working with files containing PYTHON programs. Statements to PYTHON are entered in a Windows or UNIX Shell window—the CLI—or a PYTHON Shell window—the GUI. With either interface, statements to PYTHON are structured and typed in the same way. Thus, in this chapter, we limit ourselves for the most part to describing the CLI.

5.1 Beginning a PYTHON Session

Detailed instructions for initiating a session with PYTHON will be found in the Local Guide. Usually, PYTHON will be started either (1) by typing the command ‘python’ (for the CLI) or a command like ‘idle’ or ‘python Path/idle.py’ (for the GUI) at the prompt from the operating system, (2) by double-clicking the left mouse button on an appropriate icon on the desktop, or (3) by selecting ‘IDLE (Python GUI)’ or ‘Python (command line)’ from a menu. Presently, the PYTHON prompt >>> will appear, either in a UNIX or Windows shell to the operating system (CLI) or in a newly created PYTHON Shell (GUI). We must from the outset be aware that

1. Internally, PYTHON is case sensitive. A and a do not have the same meaning.

2. Typing the statement quit() or the statement exit() at the PYTHON prompt or selecting ‘Exit’ from the File menu in the PYTHON Shell window will terminate PYTHON and return control to the operating system.

3. ⟨Control-C⟩ instructs PYTHON to abort its present activity more or less immediately. Presently, the PYTHON prompt returns and another statement can be submitted for execution.

4. Every command line in PYTHON must end either
   - with a ⟨RETURN⟩, which triggers the execution of the statement(s) on the line and, in some cases (see Section 5.3.1), the display of the result of executing the statement or
   - with the symbol ‘\’, which tells PYTHON that the statement is continued on the next line.

5. Semicolons can be used to separate independent statements on a single line.

6. In any command line, the number sign “#” can be used to introduce a comment. This sign and all following characters in the line will be ignored. Alternatively, comments occupying many lines can be enclosed in the structure “''' Comments '''” without use of the character #.

7. Within the PYTHON CLI, we can retrieve previous statements with the up-arrow key; within the Python Shell (GUI), we can retrieve previous statements by moving the cursor to the desired statement and pressing ⟨RETURN⟩. Either action will will copy the statement to the current input focus on the screen. At that point, the statement can be edited 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, the delete key to delete the character to the right of the cursor, and

---

4The PYTHON shell is defined by a PYTHON program, so bringing up the shell involves running a PYTHON program!
5Since the path will be interpreted by PYTHON, not the underlying operating system, the forward slash can be used to separate directories along the way no matter which operating system you are using.
6In these commands to the operating system, case may be important.
7Note that local configurations may have changed this prompt from the off-the-shelf default.
8The label in the PYTHON Shell may include an identification of the version in use.
9Be aware that the command that brings up the Python Shell in which you can enter statements directly may instead bring up the PYTHON Edit window. The Options menu in either of those windows offers a means to select which window will appear. See Section 5.6.
other keys to insert characters at the position of the cursor. Pressing (RETURN) will then execute the new statement, regardless of where the cursor happens to be positioned within the line.

Coupled with the multitude of third-party embellishments, PYTHON provides a very large resource for numerous tasks. Pointers to a wide variety of useful information are presented in Sections 5.8 and 5.17.

5.2 Basic Entities in PYTHON

5.2.1 Data Types

While individual items of data can be aggregated into quite complicated structures, the most frequently used data types in PYTHON are int for signed integers, float for floating point values, str for character strings, and tuple or list or set for aggregates of values (not necessarily of the same type) assigned to a single variable name. We need take no particular action to define the data type of any variable. PYTHON determines an appropriate data type automatically and dynamically. Thus, a particular variable may have one type at one point in a session and a different type at another point in the same session.

With one exception, the PYTHON commands int, float, and str, can be used to truncate a float value or convert a compatible string value to a signed integer, to convert an integer or compatible string value to a float value, and to convert an integer or float value to a string, respectively. During some conversions, e.g., the conversion of the floating point value 3.67 with the expression int(3.67), which will yield the value '3', information will be lost in the rounding and cannot be retrieved by invoking the reverse command float.

While PYTHON by default assigns a data type to each entered value, the user can force use of alternative data types through the invocation of one or another commands. For example,

```
long(a) converts an “ordinary” integer (data type int to a “long” integer (data type long). Long integers will be displayed with a suffix L.
bool(a) converts a to type bool (Boolean for logical operations with value either True or False).
hex(a) converts an integer a to hexadecimal representation of type str, and
oct(a) converts an integer a to octal representation of type str.
```

The PYTHON command type, which in the most common invocation, has a single argument, returns the type of that argument. The argument can be either an explicit value or a variable name that stores an explicit value.

5.2.2 Variable Names

The simplest variables in PYTHON represent either single numbers (type int or float), character strings (type str), or aggregates of several of these types referred to by a single name (type tuple).

---

10 A string value (e.g., ‘cook’) that does not represent a valid number cannot be converted to a number.
11 As described in Section 5.3.1, rounding floating values up or down to integers is accomplished by functions in the math module.
12 Note, however, that the distinction between “ordinary” and “long” integers is abandoned in PYTHON 3. All integers in PYTHON 3 are “long”, though data type int, not data type long, is used. The command long does not exist in PYTHON 3.
13 Any version of zero (0, 0.0, 0L, 0.0j), empty strings, empty lists, and empty tuples, and a few other values are treated as false; all other values are regarded as true.
14 The URL developer.rhino3d.com/guides/rhinopython/python-datatypes/ links to a more complete discussion of basic data types in PYTHON.
or type list or type set). Variable names must start with a letter or an underscore character, which can then be followed by any number of upper- or lower-case letters, numeric digits, and underscore characters. Except that use of reserved words like True, False, sin, and log—basically the names of built-in functions and constants—must be avoided, any name satisfying the enumerated constraints is valid. Prudence dictates that names should be chosen to have mnemonic significance and excessively long variable names, which will lead to excessively long lines of code, should be avoided. Variable names are sensitive 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.

5.2.3 Assignment of Values to Variables

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

- the statement alpha=3.79 creates a variable of type float named alpha and assigns the value 3.79 to that variable, i.e., stores in a memory location named alpha the floating point binary representation of the value 3.79.
- the statement beta=3 creates a variable of type int named beta and assigns the value 3 to that variable, i.e., stores in a memory location named beta the binary representation of the integer 3.
- the statement name = 'David' creates a variable of type str and assigns the value David to that variable, i.e., stores the codes for each of the five characters in separate adjacent memory locations.

The difference between the integer 3 and the floating value 3.0 that happens to have an integer value is significant because the two values are stored internally in different formats. Floating point values should always be entered with an explicit decimal point, even if there are no digits after the decimal point, to assure that PYTHON will adopt the proper internal storage format for that value. Note also that string constants may be enclosed in single or double quotation marks but that the returned value will most often be enclosed in single quotation marks.

Beyond setting variables equal to specific values, a variable can be set equal to an expression with the understanding that PYTHON 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 values. Thus, for example, the statement

\[
x = \text{alpha} + \text{beta}
\]

will add the values currently stored in the variables alpha and beta and store the result in the variable x, creating x if the variable does not already exist and overwriting whatever is currently stored in x if the variable already exists.

5.2.4 Commands

In PYTHON 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 PYTHON’s commands, the behavior of which is almost always influenced by the value or values of one or more arguments and/or properties. Statements invoking a particular command will usually be expressed in the form

---

15 Technically, the length of a variable name is limited only by the available memory, but testing that limit is unwise.

16 If a string contains quotation marks of one type, the entire string should be contained in marks of the other type.
5.2. BASIC ENTITIES IN PYTHON

where the arguments are separated by commas. The statement begins with the command name, which will be followed by one or more arguments, all enclosed in a single set of parentheses. Some of these arguments are “free” and others are specified by using a keyword. If a command invoked in this way is executed interactively and the command returns a value, that value will be immediately displayed on the screen in a new line.

Alternatively, those commands that return values can be invoked with a statement in the general format

\[ \text{Variable(s) } = \text{ command( argument, argument, Keyword=Value, ... )} \]

in which the value returned by the command is assigned to the user-specified variable(s)—separated by commas if plural—and the value(s) will not be displayed on the screen.

The order of the “free”—or positional—arguments is mandatory, since the position of each argument identifies its role, and all of these arguments must be provided. Arguments specified by keyword-value pairs—keyword arguments—must appear after all positional arguments but can be presented in any order, since the keyword identifies the role of the immediately following value. Most keywords have appropriate default values, so their explicit stipulation is necessary only if the default value is unacceptable. Keywords, which are always strings, are presented in the argument list without quotation marks; any values that are strings, and any positional arguments that are strings are always enclosed in single quotation marks; numeric arguments and numeric keyword values are not quoted.

5.2.5 PYTHON Modules

PYTHON has been deliberately designed with a fairly small off-the-shelf content but with a rich possibility of expanding its capabilities by the “importing” of one or more third-party or user-defined modules. To make the features of one or another module available for use in a PYTHON session in the simplest way, we simply execute either of the statements

\[ \text{import ModuleName} \quad \text{or} \quad \text{from ModuleName import *} \]

Thereafter, all components of ModuleName will be available for use. If the first statement is used, all references to components of the module must be prefaced with ModuleName; with the second statement, ModuleName can be omitted (though at some risk of confusion if components of the same name exist in two or more imported modules). The alternative statement

\[ \text{import ModuleName as Alias} \quad \text{e.g.,} \quad \text{import numpy as np} \]

provides an alias so that components can be identified by the (typically shorter) alias. For example, the numpy module array can be referred to as np.array rather than the longer numpy.array.

Further, since many modules are quite large and you may know that you need only a few components from that module, you can import only the needed components and save memory for more valuable use with statements of the form

\[ \text{from ModuleName import Component} \quad \text{e.g.,} \quad \text{from numpy import array} \]

Imported this way, the function numpy.array can be referred to simply as array, which is convenient but—WARNING—some components in a module may depend on other components in that module and one can’t be sure that components on which a particular component depends will automatically
be imported as well. Further—SECOND WARNING—confusion with identically named components in other imported modules may still arise.

There are many, many modules. Among the more useful ones, some of which must be installed separately from the main PYTHON distribution, are

- **math**, which provides numerous mathematical functions, most if not all of which are limited to accepting scalars as arguments. See Section 5.3.1.

- **numpy**, which is frequently imported with the alias `np`. This module provides numerous routines for working with arrays and also embellishes the mathematical functions provided by the `math` module so that they can accept arrays as arguments. See Sections 5.3.4 and 5.3.5.

- **matplotlib** and **matplotlib.pyplot**, which provide functions for creating a wide variety of graphical displays. See Section 5.3.9. The module `matplotlib.pyplot` is frequently imported with the alias `plt`.

The module `os`,\(^{17}\) which in particular provides functions for invoking features of the operating system from within PYTHON, and the module `sys`, which provides access to features of the underlying PYTHON interpreter, are occasionally useful and will be introduced when the need arises. The `operator` module\(^ {18}\) includes a few logical functions not included in off-the-shelf PYTHON.

### 5.3 A Sampling of PYTHON Capabilities

In this section, we present several examples illustrating various capabilities of PYTHON and introducing some of the most frequently used commands. The “conversation” in this section should start in a fresh invocation of PYTHON and will work either in the CLI or in the GUI (Python Shell).

#### 5.3.1 Using PYTHON Interactively

A simple use of PYTHON exploits its capacity to function as a sophisticated calculator. Simple arithmetic can be done simply by entering the expression to be evaluated but not assigning the result to any variable and then pressing ⟨RETURN⟩, in response to which, PYTHON will evaluate the expression and display a result. For example, after PYTHON has been launched, the “conversation” with PYTHON might unfold as follows:\(^ \text{19} \)

---

\(^{17}\)See Section 5.16.1.

\(^{18}\)See Section 5.4.

\(^{19}\)PYTHON statements are shown on the left; comments describing the statements are shown on the right. Further, to save space, we will routinely compact PYTHON’s output by omitting blank lines and extra spaces.
5.3. A SAMPLING OF PYTHON CAPABILITIES

>>> 3*4 + 2
14
>>> type(3*4+2)
<type 'int'>
>>> 8/5
1
>>> 8%5
3
>>> 8.0/5.0
0.6
>>> 2**12
4096
>>> 2**0.5
1.4142135623730951
>>> type(2**0.5)
<type 'float'>
>>> pow( 2, 12 )
4096
>>> round( pow(2,0.5), 3 )
1.414

Here, we illustrate the use of the symbols +, -, *, /, ** (or the built-in function pow), and the built-in function round for adding, subtracting, multiplying, dividing, raising to a power, and rounding numbers, respectively. Note—as mentioned also in Section 5.2.4—that these statements, which do not assign a value to a variable, result in an immediate output on a new line. Note also that PYTHON uses ** rather than ^ for raising to a power but that fractional powers are also understood.

One way to provide common mathematical functions that are not included in off-the-shelf PYTHON is to import the math module with the simple statement

>>> import math

which adds many functions, including—but certainly not limited to—exponential, logarithmic, trigonometric (forward and inverse), and hyperbolic (forward and inverse). The use of a very few of these functions is illustrated in the statements

>>> math.sqrt(9)
3.0
>>> math.fabs(-6.543)
6.543
>>> math.factorial(12)
479001600
>>> math.exp(2.5)
12.182493960703473
>>> math.sin(math.radians(90.0))
1.0

Returns square root with type float; error if argument is negative.
Returns absolute value with type float.
Returns 12!; error if argument is negative or non-integral; see math.gamma().
Returns e to the power of the argument with type float.
Returns the sine of its argument with type float; argument must be in radians. Here the function math.radians(x) converts degrees to radians.

Other available functions in the math module are math.asin(x), math.cos(x), math.acos(x), math.tan(x), math.atan(x), math.sinh(x), math.asinh(x), math.cosh(x), math.acosh(x),

We describe here the behavior of PYTHON 2. In PYTHON 3, the statement 8/5 will return 1.6, i.e., decimal points are not necessary to flag floating division. If you want an integer result in PYTHON 3, you will need to convert the floating result explicitly, e.g., int(8/5).
math.tanh(x), math.atanh(x), and math.erf(x), each of which has a single argument, and
math.atan2(x, y), which has two arguments and returns the arc tangent in the proper quadrant
determined by the separate signs of x and y. In addition, the available constants include math.pi and
math.e, each of which is provided in type float with 15 digits after the decimal point, essentially
double-precision floating point values.\textsuperscript{21}

Any of the statements in this section could, of course, be recast to assign the result of the
evaluation to a variable, e.g.

```python
>>> a = math.sqrt(9)
>>> a
3.0
```

Cast in this way, PYTHON does not automatically display the result stored in the variable used. To
display that value, one can either assert the variable name or invoke the PYTHON command print,
which can be written either with or without parentheses in PYTHON 2 but must use parentheses
in PYTHON 3.\textsuperscript{22}

5.3.2 Creating and Examining Strings, Tuples, Lists , and Sets

Entities of type \texttt{str}, \texttt{tuple}, \texttt{list}, and \texttt{set} are the simplest aggregates of the simpler entities integers,
floating numbers, and strings. These entities can be created within PYTHON or—see Section 5.6.2—
read in from files. Individual elements (or components) in strings, tuples, and lists—but not in sets—
can be accessed by specifying one or more indices to identify the location of the desired element
in the larger entity. The syntax for creating these entities and, when permitted, examining and
assigning values to individual components in these entities is illustrated in the following paragraphs.

While individual strings are normally thought of as single entities, internally they are stored as
an aggregate of individual characters. Even so, the equal sign is used to assign strings to variables.
To be properly interpreted, the sequence of characters composing the string must be enclosed in
\textit{single} or \textit{double} quotation marks, though it will most often in either case be subsequently displayed
with \textit{single} quotation marks.\textsuperscript{23} We illustrate both assigning a \texttt{string} to a variable and examining it
with the statements

\textsuperscript{21}A full listing of the functions included in the \texttt{math} module and their syntax can be found at the URL
docs.python.org/2/library/math.html or the URL docs.python.org/3/library/math.html, depending on whether
you need information for PYTHON 2 or PYTHON 3.

\textsuperscript{22}Be aware that the format of the output from a print statement depends on the data type of its argument(s)
and on whether you are using PYTHON 2 or PYTHON 3; sometimes the output will be enclosed in parentheses or
brackets, sometimes not; sometimes individual values will be separated by commas, sometimes by spaces. Be warned.
We will probably not be entirely consistent in the remainder of this book, so your output from a print statement
when replicating illustrated code may not look exactly like the output shown in this book.

\textsuperscript{23}If you need to include a single quotation mark in the string, you would write either name="don't" or name =
'don\'t'. If name is then displayed, the output will in either case be "don't" or don't, depending on whether you
simply assert the variable name or use the print command.
5.3. A SAMPLING OF PYTHON CAPABILITIES

```python
>>> name = 'David'
>>> name
'David'
>>> print( name )
David
>>> type( name )
<type 'str'>
>>> len( name )
5
>>> name[2]
'v'
```

Create a string name
Display value on the screen.
Display using print; note absence of quotation marks.
Display data type of variable.
Find number of characters in string.
Display second element in string (index of elements starts at 0).

Remember that assignment to a variable does not generate immediate output, and that simple assertion of a variable name or invocation of the function print will display the value assigned to that variable.

Creating tuples, lists, and sets, which are simple structures that more obviously than strings contain more than one element but are technically not the same nor are they vectors or arrays, involves slightly more complicated statements. To create a tuple, the list of values must be enclosed in (ordinary) parentheses and individual elements must be separated with commas. We illustrate both assigning a tuple to a variable and examining it with the statements

```python
>>> b = ( 1, 0, 3, 7, 10 )
>>> b
(1, 0, 3, 7, 10)
>>> type(b)
<type 'tuple'>
>>> b[3]
7
>>> len(b)
5
```

Create a tuple b.
Display it on the screen.
Display data type of variable.
Display third element; remember that indices begin with 0.
Display number of elements in b.

Interestingly, though individual elements in a tuple can be examined, they cannot be edited; the statement b[3]=5, for example, will generate an error message. Note that the elements of a tuple are not required to have the same type, though there may be few if any contexts in which exploiting that feature will be useful. Note also that, to create a tuple with a single element, the element must be followed by a comma, e.g., a = (4,). Repeat to be sure you notice: To create a tuple with a single element, the element must be followed by a comma.

Lists are created in the same way as tuples, but the list of values must be enclosed in (square) brackets instead of parentheses. Individual elements are still separated with commas. We illustrate both assigning a list to a variable and examining it with the statements

```python
>>> c = [ 1, 0, 3, 7, 10 ]
>>> c
[1, 0, 3, 7, 10]
>>> type(c)
<type 'list'>
>>> c[3]
7
>>> c[3] = 5
>>> c
[1, 0, 3, 5, 10]
```

Create a list c.
Display it on the screen.
Display data type of variable.
Display third element; remember that indices begin with 0.
Edit third element.
Display edited list.

In contrast to tuples, individual elements in a list can be edited. As with tuples, the elements of a list may be of different types.

---

24 See Section 5.3.4.
In a similar way, a \textit{set} can be created by replacing the enclosing parentheses in a tuple or the enclosing square brackets in a list with enclosing braces \{\ldots\}. Individual elements in a set can neither be addressed nor edited.

Even more complicated entities—lists of lists, tuples of tuples, lists of tuples, lists of lists of lists, \ldots—can be constructed. For example, a list of lists can be constructed and examined with the statements

\begin{verbatim}
>>> lstlst = [[1,2], [3,4], [5,6]]
>>> lstlst
[[1,2], [3,4], [5,6]]
>>> type(lstlst)
<type 'list'>
>>> lstlst[1]
[3,4]
>>> lstlst[1][1]
4
\end{verbatim}

Create a list of lists \textit{lstlst}. Display it on the screen. Display data type of variable. Display first element; remember that indices begin with 0. Display first element of first element.

Note that indices into entities of the sort discussed in this paragraph take the form \(1\)[1], \emph{not} \([1,1]\) or \((1,1)\).

\subsection{5.3.3 Interrogating and Adjusting the Symbol Table}

To remind ourselves of the variables PYTHON knows at any particular moment and to find out something about those variables, we might use a statement like

\begin{verbatim}
dir()
\end{verbatim}

which will return an alphabetically sorted \textit{list} of all defined variables (and will include some PYTHON-defined variables that you have not explicitly introduced). Then, should we wish to delete some of these variables to free memory for other purposes, we could use statements like

\begin{verbatim}
b = None or del b
\end{verbatim}

Here, the first statement retains the variable \textit{name} but renders the \textit{value stored therein} empty while the second removes the variable name and its value altogether. Both release some memory for subsequent use.\textsuperscript{25}

\subsection{5.3.4 Creating and Examining Arrays}

Unfortunately, because of the format of their internal storage, strings, lists, tuples, and sets are \textit{not} arrays. The off-the-shelf implementation of PYTHON, whether PYTHON 2 or PYTHON 3, does not include data types for arrays. Those components, as well as several functions and commands for manipulating them,\textsuperscript{26} are added if the module \texttt{numpy} (\textit{numerical python}) is imported with the statement\textsuperscript{27}

\begin{verbatim}
>>> import numpy
\end{verbatim}

\textsuperscript{25}Two other commands—\texttt{globals()} and \texttt{locals}—return \textit{dictionaries} (data type \texttt{dict}) of some or all defined variables, each of which is associated with its current value. These dictionaries are quite long and may be difficult to parse. The distinction between a global dictionary and a local dictionary is subtle. For purposes of being reminded of defined variables, \texttt{dir()} is sufficient.

\textsuperscript{26}And many other features, including many additional data types and many additional routines for numerical processing.

\textsuperscript{27}Many modules must be installed to complete execution of this statement, and completion may take a bit of time. See Section 5.2.5 for ways to import only those modules needed for the immediate application.
5.3. A SAMPLING OF PYTHON CAPABILITIES

Once this module has been imported, creation of an array involves invoking the command `numpy.array` which has (normally) a tuple or a list as its sole argument. For example, the statements

```python
>>> a = numpy.array( [1,2,3,4,5] )
>>> print( a )
[1, 2, 3, 4, 5]
>>> type(a)
<type 'numpy.ndarray'>
>>> a[2]
3
>>> a.dtype
dtype('int32')
```

Create a one-dimensional array `a`. Display `a`. Display data type of `a`. Note that a new type has appeared. Display second element in `a`. Remember that indices start counting at 0. Display data type of elements of `a`. Note that `int32` (32-bit signed integer), is specific to `numpy`. It does not exist in off-the-shelf PYTHON.

Create a one-dimensional array that represents a row vector and examine some of its properties. We have here encountered two numpy-specific data types (`numpy.ndarray` and `int32`) and four (data) attributes (dtype, ndim, size, and itemsize) possessed by every array.

A two-dimensional array is created by `numpy.array` from a tuple of tuples or a list of lists. For example, the statements

```python
>>> b1 = [ 1, 2, 3, 4 ]
>>> b2 = [ 5, 6, 7, 8 ]
>>> b3 = [ 9, 10, 11, 12 ]
>>> b = numpy.array( [ b1, b2, b3 ] )
```

Create a $4 \times 3$ array `b`. Display `b`. Display data type of `b`. (or `b[2][3]`) Display 2,3 element in `b`. Remember that indices start counting at 0. Also, the first index is the row index and the second is the column index.

```python
>>> type(b)
<type 'numpy.ndarray'>
>>> b[2,3]
12
>>> b.dtype
dtype('int32')
```

Display data type of elements of `b`. Note that `int32` (32-bit signed integer), is specific to `numpy`. It does not exist in off-the-shelf PYTHON.

```python
>>> [ b.ndim, b.size, b.itemsize ]
[2, 12, 4]
```

Display dimension of, number of elements in, and size in bytes of each element in `a`. Display shape (i.e., number of rows and columns) of `b`. Here, the L means that the values are long integers.

The additional (data) attribute `shape` has here subtly been introduced.

The array `b` created somewhat laboriously in the previous paragraph can be created more easily with the single statement

```python
>>> c = numpy.arange(1, 13, 1).reshape(3,4)
```

28See Section 5.3.8.
where the numpy function `arange` creates a one-dimensional array with integer elements starting at 1 (the first argument) and continuing in steps of 1 (the third argument, which defaults to 1) until the value reaches (or exceeds) the second argument.  The appended stipulation using `reshape` rearranges the 12 elements in the one-dimensional array into a $3 \times 4$ array. All three arguments of `arange` can be non-integral if necessary.

The module `numpy` also includes functions to create special arrays. Note in particular the several functions

- `numpy.zeros( [n, m] )` will create an $n \times m$ array of zeroes.
- `numpy.ones( [n, m] )` will create an $n \times m$ array of ones.
- `numpy.eye( n )` will create an $n \times n$ array with ones on the main diagonal and zeroes everywhere else.
- `numpy.random.rand( n, m )` will create an $n \times m$ array of values randomly chosen between zero and one, with a different collection generated with each successive execution. Especially when debugging a program that includes random numbers, it may be useful to generate the same sequence of random numbers with each invocation of `numpy.random.rand`. Normally, the seed for generating a sequence of random numbers is chosen arbitrarily. To assist in debugging, the module `numpy` provides the function `numpy.random.seed()` that allows the user to stipulate the seed by providing a positive integer argument between 0 and $2^{32} - 1$ (4294967295). The seed needs to be reset each time you want to restart the sequence of random numbers.

In all four cases, the elements will be floating values.

Finally, we note that `numpy` includes commands to evaluate the dot and cross products of two one-dimensional arrays. These capabilities are quickly illustrated in the code:

```python
>>> a = numpy.array( [1,2,3] )
>>> b = numpy.array( [4,5,6] )
>>> print( numpy.dot( a,b ) )
32
>>> print( numpy.cross( a,b ) )
[-3,6,-3]
```

These two commands will also return the dot and cross products of two three-element tuples and two three-element lists. Further, the command `dot` will return the dot product of one-dimensional arrays, tuples, or lists, however many elements each has (provided only that the two have the same number of elements).

Once an array has been created, we can examine individual elements in the array and edit those elements with statements like

```python
>>> a = numpy.array( [ [1,2,3], [4,5,6], [7,8,9] ] )
>>> print( a )
[[ 1  2  3 ],
 [ 4  5  6 ],
 [ 7  8  9 ]]
```

Note that, if the terminating value is exactly reached, that value will be omitted from the array.

Python 3 provides the symbol `@` to evaluate the dot product with a statement like `a@b`. 

---

29 Note that, if the terminating value is exactly reached, that value will be omitted from the array.

30 PYTHON 3 provides the symbol `@` to evaluate the dot product with a statement like `a@b`. 

---
5.3. A SAMPLING OF PYTHON CAPABILITIES

>>> a[1,2]
6

Display the element in the first row and second column of a. Remember that indices start at 0.

>>> print( a[:,2] )
[3 6 9]

Print the second column of a as a row.

>>> print( a[1,:] )
[4 6 6]

Print the first row of a.

>>> a[1,2] = 22

Edit element 1,2 and display result

>>> print( a )
[[ 1  2  3 ],
 [ 4  5 22 ],
 [ 7  8  9 ]]

We have, of course, barely scratched the surface in identifying functions and commands included in the module numpy. We shall meet other components in due course.31

5.3.5 Creating and Manipulating Matrices

Very much of computational physics makes use of vectors and matrices of various dimensions. Unfortunately, because of the format of their internal storage, strings, lists, tuples, sets, and arrays are not matrices. The numpy module also includes commands and functions for creating and manipulating matrices. We can create and display two 2 × 2 matrices with the statements

```python
>>> A = numpy.matrix( [[1,2],[3,4]] )
>>> print(A)
[[1 2],
 [3 4]]
```

Create and display matrix A

```python
>>> B = numpy.matrix( [[5,6],[7,8]] )
>>> print(B)
[[5 6],
 [7 8]]
```

Create and display matrix B

Note that commas are included in the output displayed if the variable name is asserted but are omitted if the print command is used.

With two matrices in hand, we can invoke several statements to massage them in various ways. For example, they can be added or subtracted element by element with the statements

```python
>>> A + B
matrix([[ 6,  8],
        [10, 12]])
```

```python
>>> A - B
matrix([[-4, -4],
        [-4, -4]])
```

multiplied element-by-element with the statement32

31 The URLs docs.scipy.org/doc/numpy-1.13.0/reference/ and docs.scipy.org/doc/numpy/user/quickstart.html point to more detailed enumerations and illustrations of the components in numpy.
32 Two-dimensional compatible arrays a and b can be multiplied element-by-element with the statement a*b or by invoking numpy.multiply. In either case, the result has type numpy.ndarray.
or multiplied by the rules for matrix multiplication with the statement

```python
>>> A * B
matrix([[19, 22],
        [43, 50]])
```

The two results produced from matrices will be created with data type—actually data class—`numpy.matrixlib.defmatrix.matrix`. We can evaluate the inverse of a square matrix with the statement

```python
>>> numpy.linalg.inv( A )
matrix([[3. , 2.],
        [4. , 3.]])
```

with the result having the data type of a matrix. We can evaluate the transpose of a matrix with the statement

```python
>>> numpy.matrix.transpose( A )
matrix([[1, 3],
        [2, 4]])
```

Finally, we can find the trace of a matrix with the statement

```python
>>> numpy.trace( A )
5
```

with the result having—in this case—the data type `numpy.int32`.

### 5.3.6 Complex Numbers

PYTHON uses the symbol `j` to flag a complex number. This symbol is understood even in off-the-shelf PYTHON, so the statements

```python
>>> a = 3 + 4j
>>> print(a)
(3+4j)
>>> type(a)
<type 'complex'>
>>> a.real; a.imag
3.0; 4.0
```

create and display a complex number and its data type. Complex numbers can be added (+), subtracted (−), multiplied (∗), divided (/), and raised to a power (∗∗ or `pow(...)`), including fractional powers, using off-the-shelf operations. Many of the functions in the module `numpy` also

---

33 The inverse of a square array can also be computed with `numpy.linalg.inv` but the data type of the result will be that of an array.

34 The transpose of an array can also be computed with `numpy.matrix.transpose` but the data type will be that of an array.
5.3. A SAMPLING OF PYTHON CAPABILITIES

work with complex arguments, though one must use caution. For example, `numpy.sqrt(-5)` returns `nan` (not a number) but `numpy.sqrt(-5+0j)` returns the correct value. The alternative module `cmath` (complex math) is more accommodating and will return the proper value with the statement `cmath.sqrt(-5)`. This module also has features for converting complex numbers between Cartesian and polar forms.\(^{35}\)

5.3.7 Solving Linear Equations

PYTHON’s matrix operations provide a quick route to solve 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. We would enter and verify the coefficient matrix and the vector of inhomogenities with the statements.

```python
>>> import numpy as np
>>> a = np.matrix( 
[ [2.0,5.0,3.0],
  [-1.0,3.0,-4.0],
  [1.0,-1.0,0.0]
] )
>>> print( a )
[[ 2.  5.  3.]
 [ 1.  3. -4.]
 [ 1. -1.  0.]]
```

```python
>>> b = np.matrix([ [3.0],
  [-4.0],
  [1.0]])
>>> print( b )
[[ 3.]
 [-4.]
 [ 1.]]
```

Then, recognizing in the abstract that the equation \(ax = b\), where \(a\) is a square matrix and \(x\) and \(b\) are (column) 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.

```python
>>> ainv = np.linalg.inv(a)
>>> x = ainv * b
>>> print( x )
[[ 0.85294118]
 [-0.14705882]
 [ 0.67647059]]
```

This solution correctly solves the original equations, as hand substitution—and the statement `print(a*x-b)`—will confirm.

Actually, the `numpy` module in PYTHON includes another operator that further facilitates the task of solving simultaneous linear equations. After defining `a` and `b` as above, we might simply have invoked the statement

```python
>>> print( np.linalg.solve(a,b) )
[[ 0.85294118]
 [-0.14705882]
 [ 0.67647059]]
```

---

\(^{35}\) The URL docs.python.org/?/library/cmath.html, where ? is either 2 or 3, links to a full discussion of the `cmath` module.
5.3.8 Data and Method Attributes of Variables

PYTHON variables have what are called attributes, which are accessed by following the variable name with a period (dot) and then by the name of the attribute. Attributes come in at least two types. Data attributes, more recently called instance attributes, identify information about the variable that is stored with the variable. For example, the array \( A \) defined in Section 5.3.5 has, among others, the attributes ndim, size, itemsize, dtype, and shape, all of which we met in Section 5.3.4. The values of these attributes for any particular variable that possesses the attributes can be discovered either by appending the name of the attribute to the variable name, as in the statements

\[
>>> A.ndim
2
>>> A.size
4
>>> A.itemsize
4
>>> A.dtype
dtype('int32')
>>> A.shape
(2L, 2L)
\]

Alternatively, we can exploit the built-in command getattr with the statements

\[
>>> getattr( A, 'ndim' )
2
>>> getattr( A, 'dtype' )
dtype('int32')
\]

Note the quotation marks enclosing the second argument in this function. Either single or double quotation marks can be used.

Rather than identifying a property stored as a characteristic of the variable, method attributes, which are more recently called class attributes, identify a method or a procedure that will be applied to the variable to which the attribute is attached. The statement involving the method attribute reshape used near the end of Section 5.3.4 illustrates this type of attribute.

The attributes of either type that are attached to a particular variable depend on the class and data type of the variable.

PYTHON’s features that allow users to create their own classes and attributes are beyond the scope of this book.

5.3.9 A First Graph

PYTHON’s commands also facilitate the graphing of known univariate functions, but the necessary commands are available only after the module matplotlib.pyplot has been imported. Further, we will need arrays—not tuples or lists or sets—to communicate the data to be plotted to the graphing commands. Consequently, we start by executing the statements

\[
>>> import numpy as np
>>> import matplotlib.pyplot as plt
\]
5.3. A SAMPLING OF PYTHON CAPABILITIES

the make these commands available.

To create a graph of a univariate function of a variable $x$, we need first a one-dimensional
array containing values—usually, but not necessarily, equally spaced—spanning the domain of the
independent variable from some starting point to some ending point. The necessary one-dimensional
array of can be created with a statement like

$$Var = np.arange( \text{Start}, \text{Stop}, \text{Step} )$$

To store in $Var$ a one-dimensional array of equally spaced values starting with $\text{Start}$, incrementing by
$\text{Incr}$, and stopping at the value closest to $\text{Stop}$ that does not equal or exceed $\text{Stop}$. More specifically, the statement

>>> $x = np.arange( -3, 7, 2 )$

creates the indicated array, starting at $-3$ and incrementing by 2 until the next result equals or exceeds 6. The parameter $\text{Step}$ defaults to 1 and the parameter $\text{Start}$ defaults to 0. Exploiting the defaults, the statement

>>> $x = np.arange( 5 )$

results in an array that starts at 0, increments by 1, and does not include what one might think
would be the last included value. This command returns integers if its arguments are integers and
floating values if its arguments are floating values.

When generating a list of integer indices to control a loop, the command $\text{numpy.arange}$ is
the command of choice. Alternatively (and somewhat more transparently), the desired array can be
created with a statement like

$$Var = np.linspace( \text{Start}, \text{Stop}, \text{NoVals} )$$

to store in $Var$ a one-dimensional array of $\text{NoVals}$ values starting with $\text{Start}$ and ending with $\text{Start}$. More specifically, the statement

>>> $y = np.linspace( -3, 5, 5 )$

creates the indicated array, starting at $-3$, ending at 5, and placing 5 values in that range. In this
case, the output contains both specified ending values and produces only floating values, even if its
arguments are integers. The parameter $\text{NoVals}$ defaults to 50 but $\text{Start}$ and $\text{Stop}$ must both be
provided.

With this background, suppose we want a graph of the hyperbolic cosine function $\cosh(x)$ over
the range $-3.0 \leq x \leq 3.0$. We would invoke the statements

>>> $x = np.linspace(-3.0, 3.0, 61 )$

Create a one-dimensional array with 61 elements, the first of which is -3.0, each of
whose subsequent elements differs from its predecessor by 0.1, and the last element of
which is 3.

>>> $y = np.cosh( x )$

Evaluate $\cosh(x)$ for all 61 values of $x$.

37See Section 5.4.
to set the independent and dependent variables and then the statements\textsuperscript{38}

\begin{verbatim}
>>> plt.plot( x, y )
>>> plt.show()
\end{verbatim}

to generate an internal representation of the graph and then display that graph on the screen. The
graph created by these statements appears in a new window called a \textit{Figure} window, a copy of which
is shown in Fig. 5.1. Typically, the presence of this graph on the screen will block submitting any
additional statements to PYTHON until the graph has been manually closed. Interactive plotting,
which is discussed in Section 5.15.7, is available in some PYTHON backends and permits overriding
this behavior.

Taking control of some of the defaults adopted by the command \texttt{plt.plot} or otherwise embel-
lishing the graph can be accomplished in several ways. We might, for example,

\begin{itemize}
  \item use the commands \texttt{plt.xlim} and \texttt{plt.ylim}—each of which has a single argument—a tuple
          specifying the desired ranges ($x_{\min}$, $x_{\max}$) on the $x$ axis and ($y_{\min}$, $y_{\max}$) on the $y$
          axis.
  \item use the command \texttt{plt.title} to place a title on the graph.
  \item use the commands \texttt{plt.xlabel} and \texttt{plt.ylabel} to put labels on the $x$ and $y$
          axes.
  \item use the command \texttt{plt.grid} to replace short tics along the axes with full grid lines.
\end{itemize}

Further, we might stipulate values for any of a wide variety of PYTHON \textit{keywords} to customize the
line weight and line style and to change character sizes, \ldots

Exploiting the additional commands and properties described in the previous paragraph, we
might therefore produce the more readable and useful graph shown in Fig. 5.2 with the statements\textsuperscript{39}

\textsuperscript{38}The command \texttt{plt.plot} is more fully discussed in Section 5.9.
\textsuperscript{39}Note the order of these statements. The original graph must be produced before any of the other embellishments
    can be added. Note also that, after execution of \texttt{plt.plot}, any number of additional \texttt{plt.\*} commands, including
    \texttt{plt.plot} can be executed to add components to the accumulating internal graph. Only when the command \texttt{plt.show}
    is finally executed will the accumulated components be displayed on the screen.
5.4 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. At least two such structures are available in PYTHON. The simplest is the `for` loop, which is most valuable when you know ahead of time how many times the
loop will be executed. Storing the squares of the integers from 1 to 10 in a vector, for example, is readily accomplished with the statements

```python
>>> y = [0,0,0,0,0,0,0,0,0,0]
>>> for x in range(10):
    y[x] = (x+1)**2
>>> print( y )
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Here, we have pre-allocated a 10 element list into which our loop can store the values as they are calculated. Then we write a for loop on the index x, which steps through the values in the list

```python
range(10) = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Note the colon at the end of the statement that begins with the keyword for. Note also the indentation of the statement beginning y[x]. In contrast to many computer languages (which ignore most white space), PYTHON depends critically on indentation to make clear the logical structure of code. Thus, all statements in a loop must be indented to convey to PYTHON that they are, in fact, in the loop. Since we are currently working interactively with PYTHON, we simply terminate the last statement in the loop with two pressings of the (RETURN) key to trigger the execution of the loop. The for statement steps x one at a time through all values—integers from 0 through 9, inclusive—of the loop index x, calculating the square of each plus 1 and storing each in the proper element of y. We make two further observations: (1) If the array y already exists in PYTHON’s workspace with at least 10 elements, the statement pre-allocating y can be omitted. This loop will simply install new values in the first ten elements of the existing array and will leave all other elements untouched; a safer procedure would be to invoke the statement `del y` and recreate y before executing the loop; (2) PYTHON will also correctly execute the single-line presentation

```python
>>> y = [0,0,0,0,0,0,0,0,0,0]
>>> for x in range(10): y[x] = (x+1)**2
>>> print( y )
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

In this version, only elements 0, 2, 4, 6, and 8 are computed; elements 1, 3, 5, 7, and 9 are not created.

Alternatively, we might have used the PYTHON attribute append to add an element to the end of an existing list (which may initially be empty), thereby avoiding the need to pre-allocate the entire list. For example, the coding

```python
>>> y = [0,0,0,0,0,0,0,0,0,0]
>>> for x in range(0, 10, 2): y[x] = (x+1)**2
>>> print( y )
[1, 0, 9, 0, 25, 0, 49, 0, 81, 0]
```

With an embellishment of the index, the loop can even be stepped by something other than one, as in the statements

```python
>>> y = [0,0,0,0,0,0,0,0,0,0]
>>> for x in range(0, 10, 2): y[x] = (x+1)**2
>>> print( y )
[1, 0, 9, 0, 25, 0, 49, 0, 81, 0]
```

In this version, only elements 0, 2, 4, 6, and 8 are computed; elements 1, 3, 5, 7, and 9 are not created.

If you are working in the Python Shell, indenting of the line(s) in the body of a loop will happen automatically. If you are working in the PYTHON CLI, three dots will appear at the start of each line in the loop, but you will have to insert spaces explicitly for the necessary indentation. In either case, after the last line in the loop has been typed, you will need to press (RETURN) twice to execute the loop.

If we had imported the module numpy as np, we could alternatively have used the statement `y = np.zeros10`, though that would have created an array of floating zeros.

Remember that indices in PYTHON start at 0.

Note that, even here, (RETURN) must be pressed twice after the last statement in the loop.
>>> y = []
>>> for x in range(10): y.append( (x+1)**2 )
>>> print( y )
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

achieves the same objective as the first coding above. We still have to define y as an empty list but do not need to know ahead of time how many elements will be needed.\footnote{PYTHON also admits (1) the statement \texttt{list.insert(index, value)} to insert \texttt{value} at the position \texttt{index} in an existing list, moving all following values one position later in the list and (2) the statement \texttt{list1.extend(list2)} to add a second list to the end of a first list.}

The second loop available in PYTHON is the \texttt{while} loop. Once started, a \texttt{while} loop continues executing until the condition controlling the loop changes from \texttt{True} to \texttt{False}—which means that the statements in the loop must assure that, sooner or later, that condition changes to \texttt{False}. A \texttt{while} loop that accomplishes the the same task as in the previous paragraph involves the statements\footnote{The substance of footnote \ref{footnote:while} applies also here.}

>>> y = []
>>> i = 0
>>> while i < 10:
    y.append( (i+1)**2 )
    i = i + 1
>>> y
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

Again, we pre-define \( y \) as an empty list. In addition, we this time had to initialize the loop index \( i \) and take explicit responsibility for incrementing it with each pass through the loop. 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. Note also that this loop involves two statements in the body of the loop. This \texttt{blocking} of two or more statements is communicated to the PYTHON interpreter by the indentation in the formatting of the statements.

In constructing the \texttt{while} loop in the previous paragraph, we used PYTHON’s operator \texttt{<} (less than) to express our first \textit{logical condition}. PYTHON, of course, possesses the standard six such operators, specifically \texttt{<} (less than), \texttt{>} (greater than), \texttt{==} (equal to), \texttt{<>} (not equal to), \texttt{<=} (less than or equal to), and \texttt{>=} (greater than or equal to). Finally, Boolean algebra on logical expressions is facilitated by the PYTHON operators \texttt{and} (logical and), \texttt{or} (logical or), and \texttt{not} (logical not). The logical function \texttt{xor} (logical exclusive or) is not included in off-the-shelf PYTHON but is available in the \texttt{operator} module made available with the statement \texttt{import operator} and invoked with a statement like \texttt{operator.xor(,)}, where the two arguments are logical conditions or Boolean variables.

Within PYTHON, logical conditions have values and, if set to a variable, the variable for each such condition will have type \texttt{bool}. Only two values, however, are allowed: \texttt{True} and \texttt{False}. The statements

>>> i = 5
>>> v = [ i < 10, i == 10, i > 10, i <> 10 ]
>>> v
[True, False, False, True]

reveal those possibilities.

Logical conditions appear not only in controlling loops but also in structuring branches in a sequence of statements. As with most programming languages, PYTHON also possesses
if/elif/.../else constructs, though the elif and else clauses can be omitted if they are not required. Thus, for example, the statements:

```python
>>> x = [1,-2,3,-4,5,-6,7,-8,9,-10]
>>> for i in range(10):
    if x[i] < 0.0:
        x[i] = -x[i]
>>> x
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

will replace each negative element in the ten-element vector $x$ with the corresponding positive value and the statements:

```python
>>> a = -5
>>> if a > 0:
    b = a
else:
    b = -a
>>> b
5
```

will set $b$ equal to the absolute value of $a$ (though either of the functions `math.fabs` or `numpy.abs` will do so more easily).

Finally, two reserved words are available and can play a role in structuring loops. The statement `break` results in an immediate termination of the loop, with execution resuming at the first statement after the loop. The statement `continue` omits whatever part of the loop follows the statement and re-tests the condition controlling the loop.

### 5.5 Eigenvalues and Eigenvectors

Eigenvalues and eigenvectors of two-dimensional 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 PYTHON function `eig` in the module `numpy.linalg` takes as arguments the name of the matrix whose eigenvalues and eigenvectors are desired. The function returns either the eigenvalues or the eigenvectors and the eigenvalues, depending on the structure of the variable to which its output is directed. To find both the eigenvalues and the eigenvectors, we need only create the matrix and invoke `eig` with statements like:

```python
>>> evals = np.linalg.eig(A)
```

---

47 If the lines in the body of the `for` loop are entered interactively one at a time at the PYTHON prompt in the Python Shell, the second and third lines will be automatically indented as shown here. Evidently, the prompt and its following space are not seen by the interpreter. If the lines are entered in the CLI, the characters ... will appear at the beginning of the second and third lines but the proper indentation will have to be inserted manually.

48 Here, if the lines in the body of the `if/else` structure are entered interactively one at a time at the PYTHON prompt in the Python Shell, the line `else:` will automatically start under the `b` in the line above it. To avoid an error message, you will have to remove all spaces in front of `else:` before ending the line. Even if you move the `e` of `else:` to line up under the `i` of `if`, there will be an error message. Evidently, the prompt and its following space are not seen by the interpreter, which sees the `e` as under the `i` when the `e` appears at the very beginning of the line. If the lines are entered in the CLI, the characters ... will appear at the beginning of the each line and necessary indentation will have to be inserted explicitly.

49 To find only the eigenvalues, use the statement `evals = np.linalg.eig(A)`. 
>>> import numpy as np
>>> A = np.matrix( [ [1,3,0], [3,7,2], [0,2,4] ] )
>>> print( A )
[[1 3 0]
 [3 7 2]
 [0 2 4]]
>>> evals, evecs = np.linalg.eig(A)
>>> print( evals )
[ 8.94246288 -0.38931525  3.44685237]
>>> print( evecs )
[[-0.33046479 0.89122683 -0.310657 ]
 [-0.87490144 -0.41273167 -0.25337727]
 [-0.3540346 0.18806199  0.91612892]]

Each element in \( \text{evals} \) is one of the eigenvalues of \( \text{A} \) and each \textit{column} in the array \textit{evecs} is one of the \textit{normalized} eigenvectors of the matrix \( \text{A} \), with \text{evals}[0] corresponding to \text{evecs}[:,0], ... .

The format of the output makes it easy to verify that the results are correct. We want the eigenvalues and eigenvectors to satisfy the simple equation

\[
\text{A} \times \text{x} = \lambda \times \text{x} \quad (5.2)
\]

where \( \lambda \) is an eigenvalue and \( \text{x} \) is the corresponding eigenvector. For the first eigenvalue and eigenvector, we find that\(^{50}\)

>>> lamb = evals[0]
>>> x = evecs[:,0]
>>> print( np.concatenate([A*x, lamb*x], axis=1) )

\[
[[-2.9551691 -2.9551691 ]
 [-7.82377363 -7.82377363]
 [-3.16594128 -3.16594128]]
\]

Here, evaluation of \( \text{A} \times \text{x} \) and \( \lambda \times \text{x} \) is straightforward, but arranging the two arrays in parallel columns is harder. We use \textit{np.concatenate} with keyword \textit{axis} set to 1 to combine the two columns into the desired two-column arrangement. Inspection shows that these results are the same. Similar statements focusing on the second and third eigenvalues will confirm those results as well.

Alternatively, we can verify all eigenvalues and eigenvectors at the same time. To obtain the left hand side of Eq. (5.2), we simply evaluate

>>> lhs = A * evecs
>>> print( lhs )

\[
[[-2.9551691 -0.3469682 -1.0707888 ]
 [-7.82377363 0.16068274 -0.87335404]
 [-3.16594128 -0.0732154 3.15776112]]
\]

To obtain the right hand side, however, we execute the statements

>>> diag = np.zeros([3,3])
>>> for i in [0,1,2]: diag[i,i]=evals[i]

To create a diagonal array whose diagonal elements are the three eigenvalues. We then find the right hand side of Eq. (5.2) by multiplying \textit{diag} and \textit{evecs}, but we must be careful. Because of the way matrix multiplication works, we recognize that we must evaluate the matrix product in a seemingly unnatural order. Specifically, we invoke the statements

\(^{50}\)The name \textit{lambda} is reserved in PYTHON.
to assure that each column in the result is the product of the corresponding column in \texttt{evecs} times the proper eigenvalue in \texttt{evals}. Comparison of these two matrices confirms the correctness of all three eigenvalues and eigenvectors at once. Indeed, knowing that internal arithmetic is done in double precision, we might even examine the differences between these two sides with the statement

\begin{verbatim}
>>> print( rhs - lhs )
[[ 2.66453526e-15 -6.10622664e-16 2.22044605e-16]
 [ 2.66453526e-15 2.16493490e-15 5.55111512e-16]
 [ -8.88178420e-16 -1.24900090e-16 4.44089210e-16]]
\end{verbatim}

Evidently, the two matrices, whose elements are all on the order of 1, differ by rather less than $10^{-15}$, surely equal within the roundoff errors endemic to (double precision) computer calculations.\footnote{Note that, because these results are influenced by roundoff, the actual values received may vary from platform to platform and from version to version of PYTHON. All values, however, are essentially zero.}

PYTHON’s command \texttt{eig} is quite general. The matrix on which it works must, of course, be square, but it need not be symmetric or real. Other functions in \texttt{numpy.linalg} include \texttt{eigvals}, which computes only the eigenvalues of a general, square matrix; \texttt{eigh}, which computes the eigenvalues and eigenvectors of symmetric or Hermitian arrays; and \texttt{eigvalsh}, which computes only the eigenvalue of symmetric or Hermitian arrays.

\section{The PYTHON Edit Window and Stored Programs}

For quick calculations, entering statements directly in a command window (CLI) or a Python Shell window (GUI) at the PYTHON prompt \texttt{>>>} is convenient and quick. Especially when we are doing several similar tasks or developing and debugging an extended sequence of statements, creating a command file or a program in an available text editor (i.e., creating a \texttt{script}), which can be easily edited, saved, and then executed over and over, can be an immense time saver. To facilitate that process, the PYTHON distribution includes a dedicated text editor that is linked to the program itself so that one can enter statements in the text editor and then, after saving the code in the Python Edit window, submit the statements for execution by selecting an item from a menu in the editor. The text editor is most easily launched by selecting ‘New File’ from the FILE menu in the Python Shell. Launched this way, the label in the bar at the top of the window providing the editor will be ‘Untitled’.\footnote{The FILE menu in the Python Shell window also offers both ‘Open…’, which brings up a browser in which you can search for an existing file, and ‘Recent Files’, which brings up a list of recently used Python files from which you can select the desired file. Once a named file is displayed in the Python Edit window, the label in the window will be the name of that file.}

\subsection{A Quick Example of a Stored Program}

To illustrate this way to exploit PYTHON, launch the GUI and then select ‘New File’ from the FILE menu in the Python Shell window. In the resulting Python Edit window, enter the lines

\begin{verbatim}
a = 'David'
b = 3.5
c = 100
d = ( a, b, c )
print( d )
\end{verbatim}
Then select ‘Save As’ from the File menu in the edit window, browse to a suitable directory, provide a name for the file (perhaps test), and click ML on the ‘Save’ button (or type (CONTROL-S)). Finally, select ‘Run Module’ from the Run menu in the edit window (or type (F5) on the keyboard). The line

\[
('David', 3.5, 100)
\]

will appear in the Python Shell window.

The code in the Python Edit window can be edited in any way, saved, and quickly re-executed. To illustrate, replace all of the lines in the Python Edit window with the lines

```python
y = [0,0,0,0,0,0,0,0,0,0]
i = 0
while i < 10:
    y[i] = (i+1)**2
    i = i + 1
print(y)
```

save this code by selecting ‘Save’ from the File menu (or by typing (CONTROL-S)), and execute it by selecting ‘Run Module’ from the Run menu (or by typing (F5) on the keyboard). Presently, the line

\[
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
\]

will appear in the Python Shell window. Note that, via the route described in this section, the coding you create in the Python Edit window is not mixed with the PYTHON prompt on many lines and can be saved for subsequent reading back into the Python Edit window by selecting ‘Open …’ from the File menu and browsing for the desired Python command file.

Even more convenient, a saved py-file can be executed directly from a Command window to the operating system by making the default directory the directory containing the command file and submitting the statement

```
python FileName.py
```

where python must be defined as a known command to the operating system and the file type, almost always .py, must be explicitly present. This capacity, of course, means that you can create command files in any text editor, save those files, and then execute them directly from a Command window to the operating system; using the Python Edit window is convenient but it is by no means the only editor you can use to create PYTHON programs.

A saved file can also be executed from the PYTHON prompt in the PYTHON command-line interface (CLI) or the Python Shell (GUI). After the file has been stored with a suitable name and the current directory has been set to the directory containing the file, the single statement

```
>>> execfile( 'FileName.py' )
```

in PYTHON 2 or the single statement

```
>>> execfile( 'FileName.py' )
```

---

53. Any command file called by PYTHON should be saved as a py-file, so named because the file itself by convention is named with a file type .py.
54. If python has not been defined as a legitimate command to the operating system, you will have to supply the full path to python.exe in your system.
55. Since this second route to executing a command file returns control to the operating system when the execution is complete, this route is useful only if that single command file contains a complete program.
56. See Section 5.16.1.
>>> exec(open('FileName.py').read())

in PYTHON 3 will instruct PYTHON to execute the statements in the file and ultimately produce whatever output the statements were designed to generate.

Be aware that, in contrast to many programming languages, PYTHON is an interpreted language, not a compiled language. Statements typed directly to PYTHON in the CLI or the GUI and statements input from a stored PYTHON program must be translated—i.e., interpreted—from the source code every time they are presented for execution. Even statements contained in a loop will be interpreted at each pass through the loop. This feature has advantages when developing programs but it also means that production runs of programs will require more time than would be the case if the source code could be translated once to machine code (i.e., compiled) to avoid the repeated interpretation of statements as the code is executed.

Hereafter, we shall suppress the explicit display of the PYTHON prompt and assume that all code is entered into the Python Edit window, saved, and executed by one of the methods described in this section. The statements can, of course, also be executed by copying and pasting them at the PYTHON prompt `>>>`.

5.6.2 Reading Data From a File; Radioactive Decay

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

We illustrate the process with the file `radio.dat`, which was created by a FORTRAN program that simulates the decay of a radioactive material into a material that is itself unstable. The file begins with a single line containing labels as text and continues with 201 additional lines, each containing four numerical values, each separated from the next with one or more spaces. 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. We read the data from this file with the statements

```python
import matplotlib.pyplot as plt
import numpy as np
f = open('radio.dat', 'r')
ln=f.readline()
data = []
for line in f:
    data.append(\n        [float(x) for x in line.split()]\n    )
f.close()
dataarray = np.array(data)
```

57 We assume that the file radio.dat is stored in the current directory. (See Section 5.16.1 for ways to set the current directory.) If that is not the case, then the first statement in the illustrated code must include the full path $HOME/python/radio.dat to the file. See the note at the beginning of this chapter.

58 Commas and other characters can also be used to separate values in a line. By default, the function `split` will assume that one or more spaces separate values. Any other separator should be provided as an argument to the function `split`.

59 Do not fail to copy this file into your directory and examine its structure with a text editor.

60 Here, `f.readline` reads the next as yet unread line from the file. Alternatively, the statement `f.read()` inputs the entire file, preserving the end-of-line characters and all spaces but inserting no other characters, while the statement `f.readlines()` creates a comma-separated list of the entire file, also preserving the end-of-line character and all spaces.
The first line of the file is now stored in the (string) variable ln (and—except to note that ln ends with the end-of-line character \n—we shall pay no further attention to it), and the numerical values in the file are stored in the 201 row \times 4 column array dataarray.\textsuperscript{61} We might, for example, plot the time evolution of the quantities of the three materials with the statements

\begin{verbatim}
  t = dataarray[:,0]; A = dataarray[:,1]
  B = dataarray[:,2]; C = dataarray[:,3]
  plt.plot( t, A, color='black', linewidth=3 )
  plt.plot( t, B, color='black', linewidth=3 )
  plt.plot( t, C, color='black', linewidth=3 )
  plt.grid( color='black' )
  plt.text( 2.5, 850.0, '$A$', fontsize=16 )
  plt.text( 2.5, 300.0, '$B$', fontsize=16 )
  plt.text(40.0, 800.0, '$C$', fontsize=16 )
  plt.show()
\end{verbatim}

The first two of these statements provide more compact names for each variable. The next three statements generate an internal graph of the three species. Finally, the statements invoking the command plt.text, in which locations are expressed in the coordinates of the graph, place the labels A, B, and C at appropriate points on the graph.

For easier reference, we collect the several statements composing this entire script into the lines listed in Section 5.A and store the script in a file named plotradio.py.\textsuperscript{62} Note that we have embellished this script with comments, using the character \# to introduce brief comments but enclosing more extended comments in the structure '''...'''. Several new features of PYTHON have been introduced in this example:

- The command plt.text, whose arguments use the units of the graph to specify the \((x,y)\) location of the lower left corner of the text string to be positioned at the specified point, and may include a number of properties to control the style of the displayed text. Further, we have assigned the value 16 to the property fontsize to override a somewhat smaller default size.

- The command open, which opens a file, here in read mode, and assigns to the file a variable name to be used in future reference to the file. Other modes include write mode 'w' and append 'a'.

- 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 with the method attribute readline(). The null argument reads the entire line; a numeric argument stipulates the number of bytes to be read and may yield less than a full line.

The plot resulting from executing this script by any of the means described in Section 5.6.1 is shown in Fig. 5.3.

### 5.6.3 Writing Data To a File; Formatting Output

Sometimes, data will be generated in one program but needs to be exported from that program in a way that facilitates importing the data for further processing by a different program. PYTHON provides commands for writing data from PYTHON’s memory into a file and to control the formatting of the data written into that file. To start the process, the simple statement

\begin{verbatim}
  ID = open( 'FileName.typ', 'w' )
\end{verbatim}

\textsuperscript{61}The statement print(dataarray.shape) will return \((201L, 4L)\).
\textsuperscript{62}See the note at the beginning of this chapter.
will open an empty file in the default directory, name it FileName.typ, assign to it the “handle” ID, and make it available for subsequent statements to store data in the file. Alternatively, the simple statement

\[ ID = \text{open('FileName.typ', 'a')} \]

will open an existing file named FileName.typ, assign to it the “handle” ID, and make it available for subsequent statements to append data at the end of the file. After all data have been written to the file, the simple statement

\[ ID.close() \]

will complete the creation of the file and sever its connection to the program that created it.

That is the easy part. Writing the actual data to the file in a sensible format is more complicated. We limit our discussion to the creation of text—sometimes called ASCII files—i.e., files containing printable characters, probably arranged in lines.\(^64\) To be very specific, suppose we have solved a trajectory problem for position as a function of time for some object and that we have stored in memory two one-dimensional arrays position and time, each containing 25 elements. We wish to store this information in a text file named motion.dat that starts with two descriptive lines at the beginning and then continues with 25 lines, each of which contains an entry from position and, separated from that entry by some number of spaces, the corresponding entry from time. We would begin with the code

```python
f = open('motion.dat', 'w')
f.write('Data on Motion of Pendulum
')
f.write('Generated by David Cook on 1 July 2016
')
```

\(^63\)Destroying any existing file of the specified name.

\(^64\)Writing binary files is beyond the scope of this book.
to open file for writing and write the two descriptive lines to the file. Then, to write the remaining
data to the file and close the file, we might simply use the loop

```python
for i in range(25):
    f.write( str(time[i]) + ' ' + str(position[i]) + '\n' )
f.close()
```

Here, we have recognized that the command `f.write` requires a single string as its argument. Further, we recognize that line feeds are not automatically inserted, so we must provide that stipulation with the special character `\n`.

The above coding, however, yields a file in which the values are not neatly aligned in columns because not all values have the same number of digits. We can produce a cleaner output if we format the values to have, say, 5 digits after the decimal point. If we replace the above statement starting `f.write` with the statement

```python
f.write( str(round(time[i],5))+ ' '+str(round(position[i],5))+'\n' )
```

we will arrange for all values to have no more than five digits after the decimal point, but values that do not require all five digits will not be filled with zeros, so columns in the file will still be awkwardly unaligned. To address that issue, we need to invoke explicit formatting of values by replacing the statement beginning `f.write` with the statement

```python
f.write( '{0:10.5f}{1:10.5f}'.format(time[i],position[i])+'\n' )
```

invoking format specifiers and the attribute `format`. Here, the structure `{0:10.5f}` specifies that the first argument—which with the specifier `f` must be a number—of the `format` attribute,\(^\text{65}\) as a string right-justified in a 10-character field with 5 digits after the decimal point (and will fill those digits with zeros if required), and the structure `{1:10.5f}` achieves the same objective with the second argument.

In addition to the character `f` for specifying the output of a number which has digits after a decimal point in the format `xxx.xxx`, PYTHON also provides the character `e` to output a number in scientific format `x.xxxexxx`, the character `d` to output an integer, and the character `s` to output a string. Providing output in a carefully controlled, aesthetic format can be a very complicated task. We have here described only one of numerous routes provided by PYTHON, though we have picked an approach that works in both PYTHON 2 and PYTHON 3. A full elucidation of the controls provided by PYTHON is beyond the scope of this book.\(^\text{66}\)

## 5.7 Defining Functions

### 5.7.1 Format of a Function Definition

In all programming languages, functions (sometimes called subroutines or procedures) provide an easy way to use the same code in different places in a larger code without having to duplicate the code in the listing. Functions in PYTHON have the general structure

```python
def FunctionName( arguments ):
    Statements creating the output from the inputs
    return Variables to be returned when function exits
```

\(^{\text{65}}\text{Remember that PYTHON starts counting at zero.}\)

\(^{\text{66}}\text{See the URL docs.python.org/?/tutorial/inputoutput.html, where ? is either 2 or 3, for more information.}\)
The reserved word \texttt{def} is mandatory because it informs the interpreter that the definition of a function follows, and the indentations are critical to define where the function definition ends and where the main program begins. The command \texttt{return} is necessary only if the function is to return values to the calling program. For example, the simple function

\begin{verbatim}
def sumdiff(x,y):
    sum=x+y
    diff=x-y
    return sum, diff
\end{verbatim}

will take two numeric arguments, calculate their sum and their difference, and return both values to the calling program. Once these statements have been executed to define the function, the interactive conversation

\begin{verbatim}
sumdiff(5,10)
(15, -5)
a,b = sumdiff(5,10)
a
15
b
-5
\end{verbatim}

with \texttt{PYTHON} illustrates how to use the function. The somewhat more complicated function\footnote{This function will display an error message if the roots of the quadratic polynomial are complex, i.e., if $b^2 - 4ac < 0$. A more refined function would test for and deal sensibly with that case.}

\begin{verbatim}
def quad(a,b,c):
    tmp = np.sqrt(b**2-4*a*c)
    rt1 = (b**2-tmp)/(2*a)
    rt2 = (b**2+tmp)/(2*a)
    return [rt1, rt2]
\end{verbatim}

will evaluate the two roots of the polynomial $ax^2 + bx + c$ for the coefficients supplied as arguments. Note that these statements will execute successfully to define the function, even if \texttt{numpy} has not yet been imported, but the function itself cannot be executed without prior execution of the statement

\begin{verbatim}
import numpy as np
\end{verbatim}

though \texttt{numpy} need not be imported within the function; it can be imported in the calling program and will be known globally. The statement

\begin{verbatim}
quad(1,-1,-6)
[-2.0,3.0]
\end{verbatim}

illustrates how \texttt{quad} might be called.

### 5.7.2 The Function \texttt{luplot}

Suppose we had frequent need to plot several different functions on various occasions. We might create a file named \texttt{luplot.py} containing the lines shown in Table 5.1. This coding defines the function \texttt{luplot} that plots the function \texttt{funct}, evaluating it at $N+1$ equally spaced points spanning the interval $\texttt{start} \leq x \leq \texttt{stop}$. The function itself is invoked by typing \texttt{luplot} followed by the requisite arguments in parentheses. For example, if \texttt{luplot} is stored in the current directory, the statements
5.7. DEFINING FUNCTIONS

Table 5.1: The PYTHON function luplot.py.

Program luplot.py

LUPLOT - Plots user-specified function.
Function luplot is passed a function identified by the (string) variable 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 but assigns no value to the variable fct.

```python
def luplot( funct, N, start, stop):
    x = np.linspace(stop, start, N) # Create array of independent variable
    y = funct(x) # Evaluate dependent variable
    plt.plot(x, y, 'k', linewidth=2 ) # Create internal graph
    plt.show() # Display graph
```

```python
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> execfile( 'luplot.py' )
or exec(open('luplot.py').read() )
>>> luplot( np.cos, 200, 0.0, 10.0 )
```

will 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). Note that, in this context, the name of the function in the first argument of lupplot is not enclosed in quotation marks.

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

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

for particular values of \( a \), \( b \), and \( x_0 \). We would begin by writing a py-file defining the Lorentz lineshape, perhaps using the coding listed in Table 5.2. Then, after the file lineshape.py has been stored in the current directory, we simply use statements like

```python
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> execfile( 'lineshape.py' ) or exec(open('lineshape.py').read() )
>>> execfile( 'luplot.py' ) or exec(open('luplot.py').read() )
>>> luplot( lineshape, 200, -10.0, 10.0 )
```

to define lineshape and luplot and then call luplot. In a second or two, the graph of the Lorentz lineshape shown in Fig. 5.4 will appear on the screen.

5.7.3 Using Global Variables

With the approach of the previous section, we would have to edit the function py-file if we wanted to graph the Lorentz line shape for a different set of parameters. Because of the way luplot is
Table 5.2: The PYTHON function lineshape.py.

```python
>>> Program lineshape.py

LINESHAPE - Plots Lorentz line shape.
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.

```python
def lineshape(x):
    a = 1.0
    b = 1.0
    x0 = 2.0
    y = a**2/(b**2 + (x-x0)**2)
    return y
```
variables named \( a, b, \) and \( x_0 \), which will have been assigned values at command level. Then, we invoke the already-defined \texttt{luplot} with this function using the statements

\begin{verbatim}
>>> a = 1.0; b = 1.0; x0 = 2.0
>>> execfile( 'lineshape1.py' ) or
    exec(open('lineshape1.py').read())
>>> luplot( lineshape1, 200, -10.0, 10.0)
\end{verbatim}

Assign values to global variables.
Define \texttt{lineshape1.py}
Graph function.

at the PYTHON prompt in a Python Shell window to produce again the graph shown in Fig. 5.4. This time, however, our approach is more flexible, since we can more easily change the values of the parameters and produce another graph.

The above py-files have made use of PYTHON’s capacity for self-documentation. We have again introduced short comments with the number sign \# and enclosed more extensive comments in the structure ‘‘’...’’’. In addition, for any function (user-defined or built-in) that is available at the PYTHON prompt, the simple statement

\begin{verbatim}
>>> help( FunctionName )
\end{verbatim}

where \texttt{FunctionName} is presented without quotation marks and without the file type, will display on the screen at least the line following the reserved word \texttt{def} in the function definition as a reminder of the number and nature of the arguments to be provided and, if the file is properly constructed (see PYTHON manuals), may display much more.

\textit{The wisdom of documenting user-written functions thoroughly cannot be overstressed, and this particular feature of PYTHON makes it easy to keep the documentation coordinated with—and in the same file as—the coding itself.}
5.8 On-Line Help

Several helpful resources are included in the standard PYTHON installations and can be accessed directly via commands from the keyboard:

- The statement `python --help` issued at a prompt in a command window to your computer’s operating system will display a list of options that can be appended to the command `python` followed by a list of the several environment variables that may play a role in PYTHON’s behavior, perhaps only when PYTHON is launched by the command `python` to your operating system.

- The HELP menu in the Python Shell and the Python Edit window contains three links:
  - 'About IDLE', which provides information about the version of PYTHON and the IDLE in use.
  - 'IDLE Help', which brings up a screen providing information about the content of the menus available along the top of both the Python Shell and the PYTHON Edit window.
  - 'Python Docs F1', which brings up a fully indexed and linked manual for PYTHON.

- The statement `help` issued at the PYTHON prompt `>>>` brings up a single line explaining how to use the `help` utility, either interactively or to display help on a specific topic, module, or function. For example, `help(range)` will yield information about the built-in command `range`; `help(plt.plot)` will yield information about the `plot` command in the module `matplotlib.pyplot`, but only if that module has been imported as `plt`. Typically, seeking help on a specific object via this route brings up far more detail than was sought.

- For those modules in which the attribute `info` has been provided, that attribute can be invoked to obtain more focused information. For example, if the module `numpy` has been imported as `np`, the statement like `np.info(np.sin)` produces a more informative output.

- Searching in your browser for the particular item of interest, e.g., `matplotlib.plotly.plot` will frequently bring up several links to quite specific information.

Books and resources available by searching on the web are enumerated in Section 5.17.

5.9 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 most commonly used PYTHON module for producing publication-quality graphs of all sorts is `matplotlib`.

Creating the data to be graphically displayed will almost certainly require the module `numpy` as well. Both modules must be explicitly imported into PYTHON with the statements

```python
import matplotlib
import numpy
```

These statements import all components of the modules. For our present purposes, it is more convenient to import all of `numpy` and a portion of `matplotlib` but to provide aliases by using the statements

```python
import matplotlib.pyplot as plt
import numpy as np
```

This module is often not included in the initial PYTHON distribution and must be separately installed on your computer before it can be used in PYTHON programs.

See Section 5.2.5.
import matplotlib.pyplot as plt
import numpy as np

At some point, you will learn which components you will actually need and will be able to modify
these statements to conserve memory by importing only those components that are needed.

The basic command in this module for producing a simple graph of one variable versus another
is

```python
plt.plot( x, y, Keywords/Values )
```

where \( x \) and \( y \) are one-dimensional arrays that are most conveniently created using components
present in the module `numpy`, and `Keywords/Values` abbreviates numerous optional stipulations
that can be use to modify the basic graph. The graph itself is constructed by drawing straight line
segments connecting consecutive points identified in the arrays. Both arrays must, of course, have
the same number of elements.

By using one or more of the `Keywords`, the action of `plt.plot` can be modified in a variety of
ways. The more commonly used keywords include

- `linewidth`, e.g., `linewidth=2`, where the width is measured in points, with the default value
  being 1.
- `linestyle`, e.g., `linestyle='dashed'`, with the default being a solid line.
- `color`, e.g., `color='red'`, with the default being blue.
- `marker`, e.g, `marker=’+’`, which marks individual points with the specified marker, here a plus
  sign, with the default being no marker.
- `markersize`, e.g., `markersize=12`, where the size is specified in points.

Other keywords will be introduced as the need for them arises.

Before illustrating how to produce more elaborate graphs, we present a very quick example.
The statements

```python
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace( 0.0, 10.0, 101)
y = np.sin( x )
plt.plot( x, y, color='red' )
plt.show()
```

will generate a crude graph of a sine curve. Here, we have used `np.linspace` to create an array
containing (here) 101 floating point values equally distributed in the interval \( 0 \leq x \leq 10 \) and `np.sin`
to create a second array containing values of the sine function corresponding to the values in \( x \).

The command `plt.plot` creates the graph but does not display it, so—if desired—other commands
can be used to add features to the graph before displaying it explicitly with `plt.show()`.

Be aware that, once a graph is displayed on the screen, no further commands can be submitted
to `PYTHON` until that graph has been closed.

### 5.9.1 The Basic Strategy

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

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

(5.4)

The various functions in the `math` module admit only single values as arguments. To submit an array as an
argument to these functions, we must use the corresponding functions in the `numpy` module.
Figure 5.5: On-axis magnetic field of a circular current loop.

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 PYTHON statements

```python
import matplotlib.pyplot as plt
import numpy as np
z = np.linspace(-4.0, 4.0, 101)
B = (1.0 + z**2)**(-1.5)
plt.plot(z, B, color='black', linewidth=3)
plt.title('Magnetic Field on $z$ axis', fontsize=20)
plt.xlabel('Dimensionless Position, $z$', fontsize=14)
plt.ylabel('Magnetic Field, $B_z$', fontsize=14)
plt.grid(color='black')
plt.show()
```

This sequence of statements will produce the graph shown in Fig. 5.5.\(^{71}\) We have here illustrated not only how to generate a simple graph but also how various keywords (`color`, `linewidth`, `fontsize`) can be invoked to modify the default behavior of particular statements. Note also (1) the need to extend the range in the argument of `np.linspace` by a bit so that the desired final value is included in the created array\(^{72}\) and (2) strings can include LATEX-like components that will be properly rendered in the displayed graph.

### 5.9.2 Plotting Several Graphs on One Set of Axes

Many times we may want to place several different graphs on the same set of axes. The simplest way to achieve this end is to specify the horizontal and vertical coordinates of each graph in pairs in

\(^{71}\)See Section 5.10.2 for how to export the graph to an .eps or a .pdf file for printing or incorporation in other documents.

\(^{72}\)In contrast to `np.linspace`, `np.arange`, whose arguments are (in order), the starting value, the ending value, and the increment between values—creates equally spaced values starting at the specified value but stopping at the largest value that does not actually reach the specified stopping value.
the same invocation of the command `plt.plot`. We might, for example plot superimposed graphs of the sine and cosine functions with the statements

```python
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(0.0,10.0,101)
ys = np.sin( x ); yc = np.cos( x )
plt.plot( x,ys, x,yc )
```

One useful feature of this approach is that each new superimposed graph will be displayed in a different color,\(^{73}\) so color is used to distinguish which graph is which.

Alternatively, we can use linestyle rather than color to distinguish different graphs on the same axes. For example, the plot of undamped and damped sine waves on the same axes shown in Fig. 5.6 is produced with the statements shown in Table 5.4. Here, we have illustrated use of an abbreviated way to specify the color and line style of a line in a graph without using any keywords. In the `plt.plot` statement, `'k'\(^{74}\) and `'k--'\(^{75}\) specify the line color to be black and, in the second instance, the line style to be dashed. Table 5.5 enumerates the available line styles and some of the available colors. Note that, as illustrated with `'k--'`, the abbreviations for these colors can be combined with the code for line styles. Note also that the keyword `linestyle` also is defined, so the stipulation `'k--'` could also be written `color='black',linestyle='--'`. Finally, be aware that keyword specifications must follow all non-keyword arguments, so specification of the linewidth in the above `plt.plot` command must necessarily follow all non-keyword arguments and hence will apply to all graphs in the display.

\(^{73}\)The sequence of colors can be printed on the screen by executing the statement `print plt.rcParams['axes.prop_cycle']`). Unfortunately, each color in the nine-color sequence is identified by its six-character hexadecimal value, so identifying the actual color is difficult. The first four colors are blue, orange, green, and red.

\(^{74}\)Equivalent to `color='black'`.

\(^{75}\)Equivalent to `color='black'` and `linestyle='--'`.
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(0.0,20.0,101)
sine = np.sin(x)
dampsine = np.exp(-x/10.0) * sine
plt.plot( x,sine,'k', x,dampsine,'k--', linewidth=3 )
plt.title( 'Damped and Undamped Sine Waves', fontsize=20 )
plt.xlabel('x', fontsize=14 )
plt.ylabel('$\sin(x), e^{-x/10}\sin(x)$', fontsize=14 )
plt.grid( color='black' )
plt.show()

Table 5.4: Placing two graphs on one set of axes.

Table 5.5: All line styles and some colors for use with the properties linestyle and color.

<table>
<thead>
<tr>
<th>Value for linestyle</th>
<th>Line Style</th>
<th>Value for Color</th>
<th>Color used</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>Solid (default)</td>
<td>b, blue</td>
<td>blue (default)</td>
</tr>
<tr>
<td>:</td>
<td>Dotted</td>
<td>orange</td>
<td>orange</td>
</tr>
<tr>
<td>--</td>
<td>Dashed</td>
<td>g, green</td>
<td>green</td>
</tr>
<tr>
<td>-</td>
<td>Dash Dot</td>
<td>r, red</td>
<td>red</td>
</tr>
<tr>
<td></td>
<td></td>
<td>c, cyan</td>
<td>cyan</td>
</tr>
<tr>
<td></td>
<td></td>
<td>m, magenta</td>
<td>magenta</td>
</tr>
<tr>
<td></td>
<td></td>
<td>k, black</td>
<td>black</td>
</tr>
</tbody>
</table>

5.9.3 Polar Plots and Custom Axes

PYTHON is also capable of producing polar plots and of using different axis styles instead of the standard box style. For example, to graph the cardioid defined in polar coordinates by the equation

\[ r(\theta) = a(1 - \cos\theta) \tag{5.5} \]

and shown in Fig. 5.7, we might execute the statements

import matplotlib.pyplot as plt
import numpy as np
theta = np.linspace( 0, 2*np.pi, 101 )
a = 3.0
r = a*(1.0 - np.cos(theta))
plt.polar( theta, r, linewidth=3, color='black' )
plt.title( 'The Cardioid', fontsize=20 )
plt.show()
5.9. GRAPHING SCALAR FUNCTIONS OF ONE VARIABLE

Figure 5.7: Polar plot of the cardioid

The Cardioid

increments on both axes; that feature need not be explicitly stipulated.

5.9.4 Multiple Plots In a Single Window

Sometimes we wish to plot several separate graphs in a single Figure window. PYTHON gives us substantial flexibility in formatting the layout of these graphs by providing the command subplots in the matplotlib.pyplot module. This command provides a means to control how many plots appear and where they appear in the Figure window. For a $2 \times 2$ array of plots in a single figure window, we begin by creating an empty Figure window with the command

fig, ( (ax1, ax2 ), ( ax3, ax4 )) = matplotlib.pyplot.subplots(2,2)

or, if matplotlib.pyplot has been imported as plt,

fig, ( (ax1, ax2 ), ( ax3, ax4 )) = plt.subplots(2,2)

This statement creates a Figure window and establishes names for the sub-windows in the pattern

ax1 ax2
ax3 ax4

which names will be included in statements directing output to particular sub-windows. The adjustments needed to accommodate a different pattern should be clear.

In addition, if we accept all of the defaults for the margins around the subplots and for the separations between subplots both vertically and horizontally, titles and other information surrounding the subplots may overlap the subplots. Hence, the matplotlib.pyplot module includes the subplots_adjust attribute on the name of a figure. The value of this attribute can be specified by attaching a component in the form

\[ \text{Note that the name fig and the names ax* are entirely arbitrary. They are not reserved words.} \]
\[ \text{In the present context, we need to create a subdivided Figure window before populating each subdivision with the intended display. If the display is to fill the entire window, the first execution of plt.plot will automatically create an appropriate Figure window for the display.} \]
fig.subplots_adjust( left=??, bottom=??, right=??, top=??, wspace=??, hspace=??)

to the name given to the figure that will contain the subplots. In this way, the user can specify—as a fraction of the width or height of the window—how much space is to be incorporated on the left, bottom, right, and top of the window and—as a fraction of the average axis width and height—how much space is to be left between subplots. Every parameter has a default, and only those that need adjustment need be specified.

As a quick example using this capability, suppose we wish to plot the sine and cosine functions along with their analogous hyperbolic functions. We wish to plot a $2 \times 2$ array of graphs, so we will use the `plt.subplots` command as described above and, after some trial and error, use the `subplots_adjust` attribute to modify the spacing in the display. In total, we produce the plot shown in Fig. 5.8 with the statements

```python
import matplotlib.pyplot as plt
import numpy as np
fig, ( (ax1, ax2 ), ( ax3, ax4 )) = plt.subplots(2,2)
fig.subplots_adjust(wspace=0.4, hspace=0.5)
z = np.linspace( -4.0, 4.0, 101 )
v = np.sin(z); w = np.cos(z); x=np.sinh(z); y=np.cosh(z)
ax1.plot(z,v, 'k', linewidth=3 )
ax2.plot(z,w, 'k', linewidth=3 )
ax3.plot(z,x, 'k', linewidth=3 )
ax4.plot(z,y, 'k', linewidth=3 )
ax1.set_title('Sine', fontsize=14 )
ax2.set_title('Cosine', fontsize=14 )
ax3.set_title('Hyperbolic Sine', fontsize=14 )
ax4.set_title('Hyperbolic Cosine', fontsize=14 )
ax1.set_xlabel('$z$'); ax2.set_xlabel('$z$')
ax3.set_xlabel('$z$'); ax4.set_xlabel('$z$')
ax1.set_ylabel('$\sin(z)$'); ax2.set_ylabel('$\cos(z)$')
ax3.set_ylabel('$\sinh(z)$'); ax4.set_ylabel('$\cosh(z)$')
plt.show()
```

Note (1) that we have used semicolons to separate individual statements placed on a single line and (2) that the subdivision of the Figure window is specific to the particular window here created, so that subdivision need not be explicitly undone unless the same window is to be reused in a different context.

### 5.9.5 Plotting Experimental Data

As a final example of two-dimensional graphing, we describe PYTHON’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. PYTHON 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 `rcdata.dat` containing the data was created with a text editor and stored in the current directory. 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). The two values in each line are separated by spaces.\(^{78}\) The data are read into the PYTHON workspace with the statements\(^{79,80}\)

-------------
\(^{78}\)You can, of course, copy this file from the directory `$HEAD/python` into your current directory and examine its structure with a text editor.

\(^{79}\)Make sure the default (working) directory is set to the directory containing the file `rcdata.dat`. See Section 5.16.1.

\(^{80}\)The command `numpy.loadtxt` has numerous keywords, all of whose defaults are appropriate here. More information can be found by Googling `numpy.loadtxt`.
5.9. GRAPHING SCALAR FUNCTIONS OF ONE VARIABLE

Figure 5.8: Graphs of sine, cosine, hyperbolic sine, and hyperbolic cosine on the interval $-4 \leq x \leq 4$.

![Graphs of sine, cosine, hyperbolic sine, and hyperbolic cosine](image)

Table 5.6: Some of the available symbols for use with the keyword Marker. A full listing can be found at the URL matplotlib.org/api/markers_api.html.

<table>
<thead>
<tr>
<th>Value of Marker</th>
<th>Symbol Drawn</th>
<th>Value of Marker</th>
<th>Symbol Drawn</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>point</td>
<td>v</td>
<td>triangle (down)</td>
</tr>
<tr>
<td>o</td>
<td>circle</td>
<td>`</td>
<td>triangle (up)</td>
</tr>
<tr>
<td>x</td>
<td>times sign</td>
<td>&lt;</td>
<td>triangle (left)</td>
</tr>
<tr>
<td>+</td>
<td>plus sign</td>
<td>&gt;</td>
<td>triangle (right)</td>
</tr>
<tr>
<td>star</td>
<td></td>
<td>p</td>
<td>pentagon</td>
</tr>
<tr>
<td>s</td>
<td>square</td>
<td>h</td>
<td>hexagon</td>
</tr>
<tr>
<td>D</td>
<td>diamond</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```python
import matplotlib.pyplot as plt
import numpy as np
data = np.loadtxt('rcdata.dat')

plt.semilogx( data[:,0], data[:,1], 'o', 
              linestyle='none', markerfacecolor='none', 
              markeredgecolor='black' )

plt.title('Experimental Data for High Pass Filter', 
          fontsize=20)
plt.xlabel('Frequency (Hz)', fontsize=14)
plt.ylabel('Gain (Vout/Vin)', fontsize=14)
```

Import needed modules.
Read data into two-column, 21-row floating array.
Create semilog graph, marking points with open circles. Table 5.6 shows some available symbols.
Add title and axis labels.

In the `plt.semilogx` command, the first two arguments provide one-dimensional arrays containing the horizontal and vertical coordinates of the points to be plotted, the argument 'o' stipulates marking the points with a circle, the keywords `markerfacecolor` and `markeredgecolor` select the fill and edge colors for the markers, and setting the keyword `linestyle` to 'none' suppresses the connecting of the points with straight line segments. 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. 5.9. The full program `rcfilter.py` is listed in Section 5.B.
A graph of experimental data, however, is not complete without error bars. PYTHON’s command `errorbar` is up to the task, though we need to be careful to suppress its default connecting of the points with straight lines. Supposing a symmetric uncertainty in the vertical coordinate of ±5%, we add the error bars with the statements

```
unc = 0.05*data[:,1]  # Compute uncertainties at 5%. (This array might alternatively be read from a properly expanded input file.)
plt.errorbar(data[:,0], data[:,1], unc, 
    ecolor='black', linestyle='none', 
    capsize=2 )  # Draw error bars at each point, using the calculated uncertainties as the third argument.
```

Here, we set the color of the error bars with the keyword `ecolor`, suppress the connecting of the data points with straight line segments with the keyword `linestyle`, and add horizontal bars of length 2 units at each end of each error bar with the keyword `capsize`. Note also that, once the logarithmic scale has been established with a previous invocation of `semilogx`, the logarithmic scaling will not be altered by subsequent plotting statements.

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

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

where \( f_0 \) is defined as \( f_0 = \frac{1}{2\pi RC} \) (5.6)

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.

\[
f = 10^{\text{temp}} = \left(e^{2.3026}\right)^{\text{temp}} = e^{2.3026 \times \text{temp}}
\]

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

```
r = 1.5e4  # Set value of R.
c = 0.0442e-6  # Set value of C.
f0 = 1/(2.0*np.pi*r*c)  # Calculate implied reference frequency.
temp = np.linspace( 1.0, 4.0, 26 )  # Create array with values from 1.0 to 4.0.
f = np.exp(2.3026 * temp)  # Create frequencies from \( 10^1 \) to \( 10^4 \).
gtheory = 1.0/np.sqrt(1.0 + f0**2/f**2)  # Calculate theoretical curve.
plt.plot( f, gtheory, 'k--', linewidth=2 )  # Add theoretical curve.
```

Here, we have used the values \( R = 15 \text{ kΩ} \) and \( C = 0.0442 \mu\text{F} \) and then invoked `plt.plot` to add the theoretical curve with a dashed linestyle to the accumulating internal display. Again the semilog format propagates from the earlier invocation of `plt.semilogx`.

Finally, to annotate the graph with a legend reminding us of the meaning of each of its pieces, we would use the statements

---

81 As described in the PYTHON manuals, the command `errorbar` has numerous options, including the possibility of drawing asymmetric limits and of drawing horizontal as well as vertical limits.

82 See also the function `numpy.logspace` described in the PYTHON manuals.
5.10 Making Hard Copy

5.10.1 ... of Text

To print textual results from PYTHON when PYTHON is being run in the Python Shell on an X-window device or in Windows, we can use the cutting and pasting capabilities of the X-window system/clipboard to copy the information from the window running PYTHON to the input window of the text editor in use. That file can then be edited to extract or reformat only the portions wanted. There appears to be no easily discovered way to create a file containing PYTHON output written to a Windows command window.

5.10.2 ... 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 requires two steps. First, we must create a file containing a PostScript or PDF description of the plot. Since many word processing and publishing packages have the capability to import PostScript and/or PDF files, these files are also useful when the graph is to be incorporated in a larger document.
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 PYTHON. Conveniently, the PYTHON *Figure* window has several buttons at the left end of its bottom (PYTHON 2) or top (PYTHON 3) edge. One of these buttons is labeled with a picture of an old-style small floppy disc. Clicking ML on that button will bring up a browser in which you can migrate to the desired directory and then select the desired file type—png, pdf, eps, ps, and pgf are among those available—and specify the desired name of the file. Finally, clicking ML on the button labeled 'Save' will store the contents of the *Figure* window with the specified name in the selected directory. In contrast to the behavior of some other programs, the resulting file produced in PYTHON appears to preserve the positioning of items in the display, all line widths, and all font sizes and styles.

The procedure described in the previous paragraph can be invoked only if you are interactively generating the desired display, invoking `plt.show()` to direct the display to a window on the screen, and manually carrying out that procedure. If you wish to incorporate the creation of a file containing an elaborate display by running a program in the background, that procedure is inappropriate. You can, however, replace the statement `plt.show()` in the activity that created the on-screen display with an invocation of the command `plt.savefig`. In its simplest form, we might create a file containing a crude graph of the sine function with the statements

```python
import numpy as np; import matplotlib.pyplot as plt
x = np.linspace(0, 10, 101); y = np.sin(x)
plt.plot(x,y)
plt.savefig('sinecurve.pdf')
```

The result will store the graph in a PDF file named *sinecurve.pdf* in whatever is the current directory. The command `plt.savefig` has only one “free” argument (the filename, in which the file type determines the format of the resulting file), but it also supports numerous keywords to modify the output.\footnote{\texttt{.pdf}, \texttt{.png}, \texttt{.ps}, and \texttt{.eps} are usually available.}

### 5.11 Graphing Scalar Functions of Two Variables

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

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

which defines the shape of the \([2, 3]\) mode of oscillation for a square membrane, or

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

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\footnote{See the URL \url{matplotlib.org/api/as_gen/matplotlib.pyplot.savefig.html} for a full description of these options.} two-dimensional *grid* of selected points \((x_{ij}, y_{ij})\) or \((\xi_{ij}, \eta_{ij})\) covering the portion of the \(xy\) or \(\xi\eta\) 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 \((y, \eta, \ldots)\) of those points.

\footnote{With more than one independent variable, the use of irregular grids poses a particularly imposing challenge for creating the ultimate display. We elect to limit our discussion to a regular grid of values for the independent variable.}
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 routines available in the PYTHON module numpy can process arrays element by element, a single statement, which would have the forms

\[
\begin{align*}
  z & = \text{np.sin}( 2.0\times\pi\times x ) \times \text{np.sin}( 3.0\times\pi\times y ) \\
  I & = \left( \frac{\text{np.sin}(x_i)}{x_i} \right)^2 \times \left( \frac{\text{np.sin}(\eta_i)}{\eta_i} \right)^2
\end{align*}
\]

for the functions in Eqs. (5.8) and (5.9), 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 numpy.meshgrid, plot_wireframe, plot_surface, and matplotlib.pyplot.contour.

### 5.11.1 A Preliminary: The Function meshgrid

In PYTHON, the command meshgrid in the numpy module provides the two-dimensional analog of the statement \(x = \text{numpy.linspace} (\text{Start, Stop, NoVals})\). The command meshgrid 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. Thus, for example, the statements

```python
>>> import numpy as np
>>> xx = np.linspace( -3.0, 3.0, 4 )
>>> yy = np.linspace( -1.0, 1.0, 5 )
>>> x,y = np.meshgrid( xx, yy )
```

create the arrays

```python
>>> print( x )
[[-3. -1.  1.  3.]
 [-3. -1.  1.  3.]
 [-3. -1.  1.  3.]
 [-3. -1.  1.  3.]]

>>> print( y )
[[ 0.  0.  0.  0.  0.5  0.5  0.5  0.5]
 [ 1.  1.  1.  1.  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 the single statement, say

```python
>>> z = x**2 + y**2
>>> print( z )
[[ 10.   2.   2.  10. ]]
```
Here, each entry in the array $z$ is the value of the function $z = x^2 + y^2$ 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 `meshgrid` and the nature of its output, we above chose an example that is far too small to produce a useful display of any function. We would create a more useful display if the array providing input to the several graphing routines had more entries. To prepare for subsequent examples, then, we execute the statements\(^7\)

\[
\begin{align*}
xx &= \text{np.linspace}(0.0, 1.0, 40); \\
yy &= \text{np.linspace}(0.0, 1.0, 40) \\
x, y &= \text{np.meshgrid}(xx, yy) \\
z &= \text{np.sin}(2.0*\text{np.pi}*x) * \text{np.sin}(3.0*\text{np.pi}*y) \\
xixi &= \text{np.linspace}(-3.0*\text{np.pi}, 3.0*\text{np.pi}, 50) \\
etaeta &= \text{np.linspace}(-3.0*\text{np.pi}, 3.0*\text{np.pi}, 50) \\
xi, eta &= \text{np.meshgrid}(xixi, etaeta) \\
I &= (\text{np.sin(xi)/xi})^2 * (\text{np.sin(eta)/eta})^2
\end{align*}
\]

to create more refined arrays $z$ and $I$ for the two examples of Eqs. (5.8) and (5.9).\(^8\)

### 5.11.2 Surface Plots: The Commands `plot_wireframe` and `plot_surface`

For generating an assortment of plots of functions of two variables, we need to import two additional modules, specifically.\(^9\)

```python
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

to provide plotting tools and a capacity to create projections into two dimensions of three-dimensional displays. The simplest PYTHON function from these tools for displaying two-dimensional arrays is `plot_wireframe`. The statements

```python
fig1 = plt.figure(1) 
ax1 = plt.axes(projection='3d') 
ax1.plot_wireframe(x,y,z, color='black', rstride=2, cstride=2 ) 
ax1.set_xlabel('$x$', fontsize=14) 
ax1.set_ylabel('$y$', fontsize=14) 
ax1.set_zlabel('$z$', fontsize=14) 
ax1.set_zlim( (-1.0,1.0) ) 
plt.title('[2,3] Mode', fontsize=18) 
plt.show(1)
```

will generate a three-dimensional wire-mesh surface conveying the altitudes in the two-dimensional array $z$. Here, we

---

\(^7\)We shift at this point to assuming the growing program will be created as a file in a PYTHON Edit window.

\(^8\)Note that placing an even number of points in $xixi$ and $etaeta$ avoids having the value zero in the middle of each array, thereby avoiding several divisions by zero in calculating the function $I$.

\(^9\)The module `mpl_toolkits` is included with the `matplotlib` distribution.
Figure 5.10: A wireframe representation of the function \( z \) in Eq. (5.8). The figure in (a) produced with `wireframe` is a bit confusing because the wireframe is treated as transparent, so some lines in the background actually are visible in this display. The figure in (b) produced with `plot_surface` hides the confusing lines. Note that, in the perspective shown, the \( x \) axes run positively to the east southeast, the \( y \) axes run positively to the northwest, and the \( z \) axes run positively to the north (vertical).

1. Create an empty `Figure` window
2. Add axes named `ax1` in that window with the keyword `projection` set to `'3d'` to stipulate three-dimensional data will be supplied.
3. Use `plot_wireframe` to generate the basic graph of \( z(x, y) \). Here, the arguments \( x \) and \( y \) force `wireframe` to scale the \( x \) and \( y \) axes to the physical values, and the property `color` specifies the color of the wire frame. The properties `rstride=2` and `cstride=2` specify that the wireframe diagram display only every second row (`rstride`) and every second column (`cstride`) so that the diagram is not filled too densely with lines.
4. Exploit several attributes to set a label on each axis, the limits on the \( z \) coordinate, and a title on the full display.
5. Finally, display the graph on the screen.

The resulting graph is shown in Fig. 5.10(a).

Unfortunately, because the wireframe is transparent, lines in the background are shown in and confuse this figure. To remove that confusion, we have to invoke `plot_surface` by replacing the `plot_wireframe` statement in the above code with the statement

```python
ax1.plot_surface(x, y, z, rstride=2, cstride=2, color='white', 
    shade=False, edgecolor='black', )
```

to produce Fig. 5.10(b). Here, the property `color` specifies that the surface in the figure has the same color as the background, the property `shade` suppresses lighting of the surface, and the property `edgecolor` makes the displayed edges black.

A similar coding will produce a graph of the function \( I(\xi, \eta) \) defined in Eq. (5.9). While the string `'$\xi$'` apparently is not accepted even though the string `'$\eta$'` does seem to work, we will here adopt a uniform method for generating both \( \xi \) and \( \eta \) to label the axes in the graph of \( I \).\(^{90}\) Specifically, we use the statements

\(^{90}\)See Section 5.15.1 for a more detailed discussion.
Figure 5.11: A wireframe representation of the function \( I \) in Eq. (5.9). Note that, in the perspective shown, the \( x \) axes run positively to the east southeast, the \( y \) axes run positively to the northwest, and the \( z \) axes run positively to the north (vertical).

```
import unicodedata
Gxi=unicodedata.lookup("GREEK SMALL LETTER XI")
Geta = unicodedata.lookup("GREEK SMALL LETTER ETA")

to find and assign the unicode values for \( \xi \) and \( \eta \) to the variable that will then be used in the xlabel and ylabel commands. Then, the coding

```
fig2 = plt.figure(2)
ax2 = plt.axes(projection='3d')
ax2.plot_surface(xi, eta, I, color='white', shade=False, edgecolor='black' )
ax2.set_xlabel(Gxi, fontsize=14)
ax2.set_ylabel(Geta, fontsize=14)
ax2.set_zlabel('I', fontsize=14)
ax2.set_zlim( (0.0,1.0) )
plt.title('Diffraction (Square Aperture)', fontsize=18)
plt.show(2)
```

will produce the graph in Fig. 5.11.

The axis keywords azim, specifying rotation around the \( z \) (vertical) axis, and elev, specifying the elevation angle of the \( xy \) plane, are set to angles in degrees in the view_init attribute of an axis object. These keywords allow adjustment away from the default orientation azim=-60, elev=30. For example, the statement\(^{91}\)

```
ax1.view_init( azim=30, elev=40)
ax2.view_init( azim=30, elev=40)
```

\(^{91}\)A full list of keywords and attributes is at matplotlib.org/2.0.2/mpl_toolkits/mplot3d/api.html.
changes the codes above so that they produce the reoriented views of \( z(x, y) \) and \( I(\xi, \eta) \) shown in Fig. 5.12 for comparison with Fig. 5.10 and Fig. 5.11.

Examination of Figs. 5.11 and 5.12(b) suggests that the pattern might have weak peaks (bright spots) in areas outside the main central peak. We have adopted a vertical scaling to show the high central peak. To show the secondary peaks more clearly, we could make use of the adjust the vertical scaling to specify a scale on the vertical axis that ignores the central peak. For example, we might replace the vertical scaling in the coding that produced Fig. 5.11 with the statement

\[
\text{ax2.set_zlim( (0.0,0.1) )}
\]

to 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. 5.13, reveals the secondary peaks in the diffraction pattern and even hints at some of those not on the main axes.

### 5.11.3 Contour Plots: The Command \texttt{contour}

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

\[
\text{plt.contour(x,y,z)}
\]

where the first and second arguments define the scales for the two axes while the third argument provides the data for generating the display. Exploiting a few properties, however, we could produce a more meaningful contour map with the statements

\[
\text{plt.contour( x, y, z, 20, colors='black' )}
\]

The fourth argument—here 20—specifies the number of levels to be drawn, overriding the default, and the keyword \texttt{colors} (note the 's') replaces the default coloring of contour lines and makes them all black. Evidently, when a single color is used, positive contours are drawn automatically with
solid lines and negative contours automatically with dashed lines. Otherwise, the color of each line is determined by the value of the function on that line.

Alternatively, we could replace the specification of the number of contour lines with a list giving the numerically increasing values at which contour lines are to be drawn as, for example, in the statement

```python
lns=[-0.8,-0.6,-0.4,-0.2,0.0,0.2,0.4,0.6,0.8]
plt.contour( x, y, z, lns, colors='black' )
plt.xlabel('x', fontsize=16)
plt.ylabel('y', fontsize=16)
```

Even better, we could add labels to the contour lines by modifying the contour statement and invoking clabel with the statements

```python
gr = plt.contour( x, y, z, lns, colors='black' )
plt.clabel( gr, lns )
```

Finally, we could recognize that we are dealing with a square membrane and add the statement

```python
plt.axis('square')
```

to establish a 1:1 aspect ratio for the display.\footnote{A fuller discussion of the command axis is presented in Section 5.15.4.} The end result of these several operations is the display shown in Fig. 5.14. Further details not only for the command contour but also for the related command contourf, which draws filled contour maps, can be found in the PYTHON manuals.\footnote{See the URL \url{matplotlib.org/api/plot_api.html}.}
5.11.3 Shaded Surfaces: The Command plot_surface Again

The commands plot_wireframe and plot_surface (as we used them in Section 5.11.2) display a two-dimensional array using a “wire-mesh” technique in which data points are connected to adjacent points using lines. The command plot_surface can also generate an alternate display in which a smooth surface conveys the values in a two-dimensional array. To produce this representation of the function \( z(x, y) \), for example, we might accept all defaults and execute the statements

```python
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import unicodedata
fig=plt.figure(4)
ax4 = plt.axes(projection='3d')
ax4.plot_surface( x, y, z )
```

which will produce a surface in a single color (blue by default) that also shows (faintly) some of the wireframe lines as well.

With a bit more care, we can produce a more sophisticated display. For example, we might arrange to fill in each “facet” with a color determined by the value of \( z \) at that facet. To do so, we need to include the statement

```python
import matplotlib.cm
ax4 = plt.axes(projection=’3d’)
ax4.plot_surface( x, y, z )
```

which will produce a surface in a single color (blue by default) that also shows (faintly) some of the wireframe lines as well.

With a bit more care, we can produce a more sophisticated display. For example, we might arrange to fill in each “facet” with a color determined by the value of \( z \) at that facet. To do so, we need to include the statement

```python
import matplotlib.cm
```

to provide a multitude of color maps (cm) from which to choose.\(^{94}\) We can then create a more interpretable surface plot with the statements

\(^{94}\) See the URL matplotlib.org/examples/color/colormaps_reference.html for a full enumeration of the available color maps.
Figure 5.15: Shaded surface representation of the function $z$ in Eq. (5.8).

![3,2] Mode

Figure 5.16: Shaded surface representation of the function $I$ in Eq. (5.9).

```python
ax4.plot_surface( x, y, z, cmap=cm.Greys, edgecolor='black' )
ax4.set_xlabel( '$x$', fontsize=14 )
ax4.set_ylabel( '$y$', fontsize=14 )
ax4.set_zlabel( '$z$', fontsize=14 )
ax4.set_zlim( (-1.0, 1.0) )
plt.title( '
[3,2] Mode', fontsize=18)
plt.show(4)
```

Here, the first statement replaces the simpler statement in the previous code and selects both a grey-scale color map and darkens the wire frame lines, both of which help to make the shape of the surface clearer. These statements—and nearly identical ones for the function $I(\xi,\eta)$—generate the displays shown in Fig. 5.15 and 5.16.
5.11.5 Features of the *Figure* Window

Exploration of the nature of a surface in two dimensions is greatly facilitated by several features of the *Figure* window. Note that

- Moving the cursor anywhere on the graph itself causes a message conveying the physical coordinates of the location of the cursor in the graph to appear along the right end of the bar at the bottom of the window.

- Pressing and holding ML when the cursor is somewhere on the display picks up the display and permits reorientation of the graph (including axis labels but not titles) in real time by moving the mouse. When the mouse button is released, the display is fixed at its new perspective. Note that, when the mouse button is being held, a pop-up message at the right of the bottom edge displays the azimuth and elevation of the current orientation—values that could well be useful in the subsequent construction of a statement exploiting the command `view_init`—See the text associated with Fig. 5.12—to produce the identical view from the command line.

- Pressing and holding MR when the cursor is somewhere on the display and moving the cursor effects zooming of the graph.

- Pressing and holding ML when the cursor is somewhere on the display and moving the cursor effects panning of the graph.

- Clicking ML on the left-most button along the bottom of the *Figure* window will restore the original view after other actions have “disturbed” it.

- Clicking ML on the button showing a magnifying class and then drawing a rectangle on the graph while holding ML will zoom to the rectangle when ML is released.

- Clicking ML on the button showing a floppy disk will bring up a window in which you can provide information for saving the display in a file of any of several types. (See Section 5.10.2.)

5.11.6 Functions of Two Variables in Polar Coordinates

Graphing a function \( z = f(r, \phi) \) defined in polar coordinates involves evaluating the function on a grid of polar coordinates but then converting the \( r \) and \( \phi \) coordinates to Cartesian values before invoking plotting procedures already described. If, for example, the function we wish to plot is

\[
z = (1 - r^2)e^{-r}\cos(\phi)
\]  

(5.10)

and we sought a graph over the intervals \( 0 \leq r \leq 5.0 \), \( 0 \leq \phi \leq 2\pi \), we would begin by importing the necessary modules, defining the independent variables, creating the two-dimensional grids, and evaluating the function with the statements

```python
import numpy as np
import matplotlib.pyplot as plt
rr = np.linspace(0.0, 5.0, 25)
phiphi = np.linspace(0.0,2.0*np.pi, 36)
r, phi = np.meshgrid(rr, phiphi)
z = (1-r**2)*np.exp(-r)*np.cos(phi)
```

Next, we would convert the polar coordinates to their Cartesian equivalents with the statements

```python
x = r*np.cos(phi); y = r*np.sin(phi)
```
Finally, for a crude graph, we complete the task of generating a wire frame graph with the statements

```python
fig = plt.figure()
ax = plt.axes(projection='3d')
ax.plot_surface(x, y, z, color='white', edgecolor='black')
plt.show()
```

The result is shown in Fig. 5.17. Adding labels and a title is left to the reader.

A similar procedure will create a polar contour plot. We simply replace the last four statements above with the three statements

```python
fig = plt.figure()
plt.contour(x, y, z, 20, colors='black')
plt.show()
```

to produce the display shown in Fig. 5.18.

The purpose of this brief section is to illustrate the strategy. You are on your own to embellish it as needed in your context.

### 5.12 Graphing Scalar Functions of Three Variables

Scalar 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,
Figure 5.18: Contour map of the function $z$ in Eq. (5.10).

Each of whose entries 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. This probability density is given as a function of Cartesian coordinates $(x, y, z)$ with the nucleus located at the origin by

$$p(x, y, z) = \frac{1}{2\pi(27)^{3/2}} e^{-2\rho/3} \left( \frac{3z^2}{\rho^2} - 1 \right)^2 = \frac{1}{2\pi(27)^{3/2}} e^{-2\rho/3} (9z^4 - 6z^2\rho^2 + \rho^4) \quad (5.11)$$

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

$$\rho = \sqrt{x^2 + y^2 + z^2} \quad (5.12)$$

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 32 segments, which will entail evaluating $p(x, y, z)$ over a grid containing 33 values of $x$, 33 values of $y$, and 33 values of $z$—a total of $33 \times 33 \times 33 = 35937$ points.

Using PYTHON to generate a three-dimensional array containing values of this probability density at a grid of points in the desired region is straightforward, since the command \texttt{meshgrid} can accept not only two ranges (to produce two two-dimensional arrays) but also three ranges (to produce three three-dimensional arrays). An array containing values of the probability density in Eq. (5.11) is created using the statements

```python
import numpy as np
xx = np.linspace(-10.0, 10.0, 33)
yy = np.linspace(-10.0, 10.0, 33)
zz = np.linspace(-10.0, 10.0, 33)
x, y, z = np.meshgrid(xx, yy, zz)
pfactor = 1.0/(2.0*np.pi*27.0**3)
zs = z**2
rhos = x**2 + y**2 + zs
rho = np.sqrt(rhos)
p = pfactor * np.exp(-2.0*rho/3.0) * \
(9.0*zs**2 - 6.0*zs*rhos + rhos**2)
```

Import numpy.
Calculate grid.
Calculate premultiplier.
Calculate $z^2$.
Calculate $\rho^2$.
Calculate $\rho$.
Calculate $p(x, y, z)$.
The first statement provides numerical routines, the next four statements establish the grid, and the remaining statements lead ultimately to an evaluation of the function \( p(x, y, z) \) whose values are stored in the three-dimensional array \( p \), which will provide the input for the production of a variety of displays.

### 5.12.1 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, \( z = z_0 + 5 \times dz = -10.0 + 5 \times 20.0/32.0 = -6.875 \)). Thus, the statements

```python
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm

fig1 = plt.figure(1)
ax1 = plt.axes(projection='3d')
ax1.plot_surface(x[:,:,5],y[:,:,5],p[:,:,5], color='white', shade=False, edgecolor='black' )
plt.show(1)

fig2 = plt.figure(2)
ax2 = plt.axes(projection='3d')
ax2.plot_surface( x[:,:,5], y[:,:,5], p[:,:,5], cmap=cm.Greys, edgecolor='black' )
plt.show(2)

fig3 = plt.figure(3)
plt.contour( x[:,:,5], y[:,:,5], p[:,:,5], 20, colors='black' )
plt.axis( 'square' )
plt.xlabel('x', fontsize=16); plt.ylabel('y', fontsize=16)
plt.show(3)
```

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

\[95\] We must extract two-dimensional arrays from all three arguments, since these statements will not behave properly with three-dimensional arrays in any position.
Figure 5.19: Wireframe representation of the function \( p \) in Eq. (5.11) in the plane \( x = x_0 + 15 \times dx = -10.0 + 15 \times 20.0/32.0 = -0.625 \).

```
px = np.zeros((33,33))
for i in np.arange(33):
    for j in np.arange(33):
        px[i,j] = p[15,i,j]
yyy = np.linspace(-10.0,10.0,33)
yy,zz = np.meshgrid(yyy, yyy)
fig4 = plt.figure(4)
ax4 = plt.axes(projection='3d')
ax4.plot_surface(yy, zz, px, color='white', shade=False, edgecolor='black')
ax4.set_xlabel('y', fontsize=14)
ax4.set_ylabel('z', fontsize=14)
ax4.set_zlabel('p', fontsize=14)
plt.show(4)

fig5 = plt.figure(5)
plt.contour(y[:,:,0], z[:,:,0], px, 20, colors='black')
plt.xlabel('y', fontsize=14)
plt.ylabel('z', fontsize=14)
plt.axis('square')
plt.show(5)
```

Preallocate storage for array.
Extract data in plane parallel to \( yz \) plane.
Create arrays for scaling axes.
Generate and label wireframe plot.
Generate and label contour plot.

for example, will extract data in a plane parallel to the \( yz \) plane and generate first a wireframe surface and then a contour map of those data. The figures produced by these commands are presented in Fig. 5.19 and Fig. 5.20.

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 a small portion of the data.
Figure 5.20: Contour map of the function $p$ in Eq. (5.11) in the plane $x = x_0 + 15 \times dx = -10.0 + 15 \times 20.0/32.0 = -0.625$.

5.12.2 Isosurfaces

Another method of viewing a three-dimensional scalar field is to look at isosurfaces, or sets of locations at which the function has a constant value. Both the mayavi module and the plotly module include resources for producing isosurfaces and may provide resources for displaying contour maps in planes representing oblique slices through volumetric data. Both of these modules are quite involved and, of course, must be installed—usually separately from the PYTHON installation—before their features can be invoked. For the sake of a simple illustration, we here assume that the plotly module and the supporting skimage module have been installed on your platform, and we limit ourselves to creating an isosurface within the data generated in the variable $p$ defined in Eq. (5.11) and already created in the Python Shell at the beginning of Section 5.12.96 We begin by importing the necessary modules with the statements

```python
import plotly as pl
import plotly.figure_factory as ff
from skimage import measure
```

Then, we calculate the vertices, faces, and other aspects of the isosurface at the value $p = 1.2 \times 10^{-4}$ and extract the coordinates of the vertices that define the isosurface with the statements

```python
verts, faces, normals, values = measure.marching_cubes_lewiner( p, level=1.2e-4 )
x1,y1,z1 = zip(*verts)
```

Here, the asterisk in the argument of the command `zip` stipulates that an already zipped variable is to be unzipped. Next, we create the internal description of the display with the statement

```python
fig = ff.create_trisurf(x1, y1, z1, faces, title="Isosurface", 
                        show_colorbar=False, colormap=['rgb(255,255,255)'])
```

96Refer to the URL plot.ly/python/isosurfaces-with-marching-cubes/ for full detail on the plotly module.
Here the stipulated value of the keyword colormap specifies that the isosurface itself be displayed in white; the lines that help define the surface will by default be shown in black. Finally, we display the result with the statement

\[
\text{pl.offline.plot(fig, filename='c:/users/cookd/trial.html')}\]

which, because of the offline stipulation, writes the display into the html file specified on your local computer (rather than to your account in the cloud) and automatically displays that file in your default browser.

Unfortunately, the above described process labels the axes with the indices of the values in the \(33 \times 33 \times 33\) in the original arrays \(x, y, \) and \(z\), i.e., with values in the range \(0.0 \leq x, y, z \leq 32.0\). Physically, the range of these variables was \(-10.0 \leq x, y, z \leq 10.0\). One way to obtain the correct labeling involves merely rescale \(x_1, y_1, \) and \(z_1\) to span this new range with statements like

\[
x_2 = \text{np.array}(x_1); \quad x_{11} = 0.625*x_2 - 10.0
\]
\[
y_2 = \text{np.array}(y_1); \quad y_{11} = 0.625*y_2 - 10.0
\]
\[
z_2 = \text{np.array}(z_1); \quad z_{11} = 0.625*z_2 - 10.0
\]

With this change, executing the two statements

\[
\text{fig = ff.create_trisurf(x_{11}, y_{11}, z_{11}, faces, title="Isosurface", show_colorbar=False, colormap=["rgb(255,255,255)"])}
\]
\[
\text{pl.offline.plot(fig, filename='c:/users/cookd/trial.html')}\]

produces the desired isosurface with more appropriate labels on the axes. Note that the image on the screen can be reoriented by pressing and holding ML with the cursor somewhere within the display and moving the mouse around.

The display resulting from the above commands is shown in Fig. 5.21. Note that, to incorporate this display, originally captured in an .html file, into a L\(\TeX\) document, we display the graph on the screen, use a screen-copy tool to create a .jpg file of the graph, and exploit the ability of L\(\TeX\) to include .jpg files in the \texttt{\includegraphics} command.

This section has provided the skimpieest of orientation to using plotly for creating isosurfaces. Anything more is beyond the scope of this book.

5.13 Graphing Vector Fields

Graphical display of vector functions of two or three variables is more complicated than display of scalar functions, but PYTHON has at least one function 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. 5.22 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 \(E\) and the magnetic field \(H\) are given at time \(t = 0\) by

\[
E_z = \cos \kappa_x b \sin n \pi y
\]

\[97\]The coefficients in these linear transformations were obtained by solving the equations \(-10.0 = 0.0 * a + b\) and \(10.0 = 32.0*a + b\) by the method described in Section 5.3.7. Note that converting \(x_1, y_1, \) and \(z_1\) to arrays is necessary before the linear rescaling can be achieved.

\[98\]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 for awhile after January, 2003, in a Dover reprint, but that reprint has since not been kept in print.
Figure 5.21: An isosurface of the probability density $p$ in Eq. (5.11) using components of the plotly and skimage modules.

Figure 5.22: A waveguide with rectangular cross section.
\[ H_x = \sin \kappa \frac{b\pi}{n} \cos n\pi \bar{y} \]  \hspace{1cm} (5.13) \\
\[ H_y = -\kappa \frac{b}{n\pi} \cos \kappa \frac{b\pi}{n} \sin n\pi \bar{y} \]

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, \( \bar{x} = x/b \) and \( \bar{y} = y/b \) are dimensionless coordinates within the guide, and

\[ \left( \frac{\kappa \frac{b}{n\pi}}{n} \right)^2 = \left( \frac{\omega b}{n\pi c} \right)^2 - 1 \]  \hspace{1cm} (5.14)

where \( c \) is the speed of light and \( \omega \) is the frequency of the wave. The field depends on two parameters, \( n \) and \( \kappa \), where \( \kappa \) is determined from Eq. (5.14) by \( \omega \), \( b \), and \( c \). Since \( \kappa \) 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 \( \kappa b/2\pi \) turns out to have the value 1. With these choices, the fields are given by

\[ E_z = \cos 2\pi \bar{x} \sin 2\pi \bar{y} \]
\[ H_x = \sin 2\pi \bar{x} \cos 2\pi \bar{y} \]
\[ H_y = -\cos 2\pi \bar{x} \sin 2\pi \bar{y} \]  \hspace{1cm} (5.15)

where \( \bar{x} \)—the coordinate along the guide 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. These arrays are readily created with the PYTHON statements\(^\text{90}\)

```python
import numpy as np
import matplotlib.pyplot as plt
xx = np.linspace(0.0,1.0,26)
yy=np.linspace(0.0,1.0,26)
x, y = np.meshgrid( xx, yy )
xp = 2.0*np.pi*x; yp = 2.0*np.pi*y
Ez = np.cos(xp) * np.sin(yp)
Hx = np.sin(xp) * np.cos(yp)
Hy = -np.cos(xp) * np.sin(yp)
```

### 5.13.1 The Function quiver

The PYTHON function `quiver` produces a two-dimensional vector field plot. At each grid point, `quiver` draws an arrow which conveys both the direction and the magnitude of the field at that point. The calling statement for this function is

```python
plt.quiver( x, y, u, v )
```

where \( x \) and \( y \) are either one-dimensional arrays containing the actual coordinates of the grid points on the two axes or two-dimensional arrays containing the coordinates of all points in the grid at which vectors are to be drawn, and \( u \) and \( v \) are the \( x \) and \( y \) components of the two-dimensional field, respectively, at the grid points conveyed by \( x \) and \( y \). Thus, for example, the two components of \( \mathbf{H} \) in PYTHON’s memory can be displayed in this form by executing the statement

\(^{90}\text{From here on in this section, we drop the overbars.}\)
Figure 5.23: Magnetic field of Eq. (5.15) at $t = 0$ in a rectangular wave guide using quiver. The illustrated structure, of course, propagates along the guide toward positive $x$ as time unfolds.

```python
fig1 = plt.figure(1)
plt.quiver( x, y, Hx, Hy )
plt.xlabel( '$x$', fontsize=14 )
plt.ylabel( '$y$', fontsize=14 )
plt.show(1)
```

The resulting display is shown in Fig. 5.23.

### 5.13.2 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 quiver are superimposed on a contour map of the $z$ component of the electric field. The statements

```python
lns = [-0.9,-0.7,-0.5,-0.3,-0.1,0.1,0.3,0.5,0.7,0.9]
plt.contour( x, y, Ez, lns, colors='black' )
plt.title('Magnetic Field in a Rectangular Wave Guide', fontsize=16 )
```

will add contour lines to Fig. 5.23, producing the display shown in Fig. 5.24. Here, we have used arguments to contour to specify that contours for positive values should be drawn with a solid line (the default linestyle) while contours for negative values should be drawn with a dash-dot line (linestyle ‘-.’). 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 dash-dot line.
Figure 5.24: Plot showing both magnetic and electric fields of Eq. (5.15) 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 whose magnitude is conveyed by the contour lines. The illustrated structure, of course, propagates along the guide toward positive $x$ as time unfolds.

5.13.3 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 `plt.quiver` to draw pictures of the projections of the field in any given plane parallel to a coordinate plane into that plane. In addition—and more usefully—if one executes the statements

```python
from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
import numpy as np

xx = np.linspace(-1.0,1.0,6)
yy=np.linspace(-1.0,1.0,6)
zz=np.linspace(-1.0,1.0,6)
x, y, z = np.meshgrid( xx, yy, zz )
r2 = x**2 + y**2 + z**2; r3 = r2**1.5
Ax= x/r3; Ay=y/r3; Az=z/r3

fig = plt.figure()
ax = plt.axes(projection='3d')
ax.view_init( azim=-41, elev=22)
```

Figure 5.25: A three-dimensional vector field.

```python
ax.quiver(x, y, z, Ax, Ay, Az, length=0.2, normalize=True)
plt.show()
```

will evaluate the vector field

$$\mathbf{V} = \frac{\mathbf{r}}{r^{3/2}}, \quad \mathbf{r} = x \mathbf{i} + y \mathbf{j} + z \mathbf{k}$$

(5.16)

of a point electric charge over the interval $-1 \leq x, y, z \leq 1$ and produce the on-screen display shown in Fig. 5.25. Viewed on the screen, this display can be reoriented by dragging it with ML to help visualize the three-dimensional vector field. Further exploration of these capabilities is left to the exercises.

## 5.14 Animation

### 5.14.1 The Basic Command

Along with all of its computational and graphing capabilities, PYTHON can also display a sequence of images in rapid succession to produce an animated display. In essence, one creates and stores each image in turn and then displays each of those images one at a time. For producing animations from a succession of images, the important function is imported and invoked with the statements

```python
from matplotlib import animation
anim = animation.FuncAnimation(fig, func, frames=Integer, init_func=InitFuncName, interval=Integer, repeat=TrueOrFalse)
```

where

- `fig` names the *Figure* window created by `plt.figure()` and in which each frame will be displayed.
• *func* identifies a function that returns the graphic display for the next image in the animation. Its first argument must be an integer variable that numbers the images. If needed for the execution of the function, further arguments can be added.

• *frames* is (most often) an integer specifying the number of frames to be included in the animation. Other possibilities are described at the URL following this bulleted list.

• *init_func* identifies a function that sets the initial image. It is optional and, if it is omitted, the initial image will simply be the first image provided by *func*.

• *interval* specifies in milliseconds the delay to be introduced after each image is displayed (and provides a means to prevent the animation from unfolding too rapidly). It defaults to 200. The amount of computation needed to create each image may impose an upper limit on the speed at which images can be displayed.

• *repeat* is a logical variable. It defaults to *True*. If *False*, the succession of images will be shown once. If *True*, the succession will be endlessly repeated.

Further description of this function and several additional keywords can be found at the URL matplotlib.org/api/as_gen/matplotlib.animation.FuncAnimation.html.

### 5.14.2 Motion of a Vibrating String

As a first illustration, we create an animated display showing the vibration of a flexible string oscillating in one of its harmonics. The function

\[ u(x, t) = \cos(\omega t) \sin(n \pi x) \]  

(5.17)

describes that motion. We begin by executing the statements

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import animation
```

```python
fig = plt.figure()
ax = plt.axes(xlim=(0,1), ylim=(-1.5,1.5))
ax.grid()
line, = ax.plot([],[],linewidth=2, color='black')
```

thereby assuring that the scaling will remain the same in each generated frame and a grid will be included in each frame. The last line here creates a variable to anticipate the successive creation of the data points to be plotted for each frame, preparing for later insertion of data by providing two empty lists with the empty square brackets and stipulating the desired linewidth and color.

We also need to define two functions—see Section 5.7—that will be provided as arguments to the routine that creates the animation. One of these functions, specifically

```python
def init():
    line.set_data([],[])
    return line,
```

will be used to initialize the variable line as creation of each image in the animation is started. The other, specifically
def animate(i, harm):
    x = np.linspace(0.0, 1.0, 101)
    y = np.cos(i*np.pi/50.0)*np.sin(harm*np.pi*x)
    line.set_data(x, y)
    return line,

will be invoked to provide the data for each line in turn. Here, the names init and animate are entirely arbitrary, and the argument i to the function animate records the frame number at each step. Further, the variable harm will be assigned an integer value in the main program and will influence here the harmonic whose animation is generated when the program is run. The variable harm is defined in the main program and passed through FuncAnimation to animate by using the keyword fargs.

With the above preparations, we are ready to invoke the command animation.FuncAnimation that will generate and store the several frames. In the present context, we select a particular harmonic\textsuperscript{100} and create the animation with the statements

\[
harm = \text{int}(\text{input('Specify harmonic: ')})
\]
\[
\text{anim} = \text{animation.FuncAnimation(} \text{fig, animate, init_func=init,} \text{frames=100, interval=20, repeat=True, fargs=(harm,)})
\]

Finally, the statement plt.show() will create the actual animation on the screen. This animation will run freely until it is stopped by clicking ML in the large \(\times\) in the upper right corner of the Figure window.\textsuperscript{101}

The full listing of the program stwave.py is appended in Section 5.C. The program can be executed by loading it into the PYTHON Edit window and selecting 'Run Module' from the Run menu or typing (F5). Alternatively, provided the current directory is properly set in each case, the program can be run from within the Python Shell with the statement

\[
\text{execfile('stwave.py')} \quad \text{or} \quad \text{exec( open('stwave.py').read() )}
\]

or from a command window to the operating system with the statement

\[
\text{python stwave.py}
\]

However the program is launched, it asks for the entry of a harmonic number before generating the animated display.

5.14.3 Motion of an L-Shaped Drumhead

For a more complicated illustration, we use a data file containing forty-one 33 \(\times\) 33 arrays, each of which conveys the displacement of an L-shaped membrane at a particular moment in time.\textsuperscript{102} Starting at the beginning of the file, each group of six consecutive lines contains the 33 values in one row of the 33 \(\times\) 33 array. We begin by importing necessary modules with the statements

\textsuperscript{100}Explicit conversion of the value returned by the input command works in both PYTHON 2 and PYTHON 3, and is necessary in PYTHON 3. In PYTHON 2, string entries must be enclosed in quotation marks but integers, floating values, and Boolean values will be dynamically and properly typed. In PYTHON 3, all values are stored initially as strings and must be explicitly converted if necessary.

\textsuperscript{101}See Section 5.16.6 for a way to trap the input of an invalid value for harm and offer the user a second chance. Otherwise, an invalid entry will crash the program.

\textsuperscript{102}The file was created by using lsode—the Livermore solver for ordinary differential equations—to effect a numerical solution of the wave equation for the membrane.
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import animation

Then, we read in the data with the statements

```python
f = open( 'ldrum.dat', 'r' ) # Open file for reading.
data = [] # Initialize variable for data
for line in f: # Read each line containing six
    # values, converting each to
    # a number.
    data.append( [ float(x) for x in line.split() ] )
f.close() # Close file
```

At this point, the structure of the file data consists of 1353 groups, each of which contains six lines in the form

```
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
[0.0, 0.0, 0.0]
```

Each such group contains the 33 values that must be concatenated to construct a single row in one of the $33 \times 33$ arrays describing the shape of the membrane at a single instant, and 33 of these rows must then be assembled to construct the array itself. The coding

```python
i = 0
k = 0
drm=np.zeros( (41,33,33) )
while k <= 40:
    tmp = []
    j = 0
    while j <= 32:
        nxtln = np.concatenate([data[i],data[i+1],data[i+2], \
                               data[i+3],data[i+4],data[i+5]], axis=0)
        tmp =np.concatenate( [tmp, nxtln], axis=0 )
        i=i+6
        j=j+1
    drm[k] = np.reshape(tmp, (33,33) )
    k = k+1
```

ultimately constructs the three-dimensional array drm that contains forty-one $33 \times 33$ arrays providing the input for the animation we seek. The inner while loop controlled by the index j will be executed 33 times. In each pass, six lines are read from the array data and concatenated into a one-dimensional array named tmp (which is initialized outside the loop). On exit from this inner loop, tmp contains $33 \times 33 = 1089$ values which are reshaped to a $33 \times 33$ array and stored in the element drm[k][:][:] by the statement that immediately follows that loop. Finally, this whole structure is placed inside the outer while loop controlled by the index k. When this coding has finished executing, we have stored in the variable drm forty one $33 \times 33$ two-dimensional arrays constructed from the data in the original file ldrum.dat.
Beating the data the original file into a form that facilitates generating the animation has taken some effort. To complete the task, we need merely initialize the axes for the animation to come with the statements:

```python
xx = np.linspace(0.0, 1.0, 33); yy = np.linspace(0.0, 1.0, 33)
x, y = np.meshgrid(xx, yy)
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.set_xlim((0.0, 1.0))
ax.set_ylim((0.0, 1.0))
ax.set_zlim((-0.25, 0.25))
def surf(k):
    ax.clear()
    ax.set_zlim((-0.25, 0.25))
    pic = ax.plot_surface(x, y, drm[k], color='white', edgecolor='black')
    return pic,
```

define the function required by the animation routine with the statements:

```python
anim = animation.FuncAnimation(fig, surf, frames=41, repeat=True)
plt.show()
```

The full listing of the program `drumhead.py` is appended in Section 5.D. Provided the current directory is properly set in each case, the program can be run from within the Python Shell with the statements:

```python
execfile('drumhead.py')
xexec(open('drumhead.py').read())
```

or from a command window to the operating system with the statement:

```python
python drumhead.py
```

### 5.14.4 Trajectory of a Projectile in Uniform Gravity

In addition to animating the time evolution of functions of one and two independent variables (plus time) as illustrated in Sections 5.14.2 and 5.14.3, one sometimes wants to animate a display of the motion itself of some physical system. PYTHON has rich capabilities to respond to that desire. Because this capability is not a major focus of this book, however, we here merely whet your appetite by laying out a very simple example showing the motion of a projectile fired at an angle up from the horizontal. Suppose the projectile starts at \((x,y,z) = (0.0, 0.0, 0.0)\) m and is fired with initial velocity \((v_x, v_y, v_z) = (v_{x0}, v_{y0}, 0.0)\) m/s. Until it hits the floor, its position relative to its starting point is then given as a function of time \(t\) by

\[
x(t) = v_{x0} t ; \quad y(t) = -\frac{1}{2} gt^2 + v_{y0} t ; \quad z(t) = 0.0
\]

(5.18)

The motion continues as long as \(y(t) > 0\).

---

103 We place this definition at the beginning of the file.
104 Full details can be found by going to the URL `vpython.org` and selecting 'Documentation' in the resulting screen.
Since, for this example, we are working in PYTHON 3.6.5 using VPYTHON 7.4.6, we construct our animation from components in the vpython module, which must be installed in your system before it is available for use. While more advanced commands allow you to create and name the canvas on which your animation is to be drawn, a default canvas named scene is created automatically. Note also that, while your program is run as always in PYTHON, the display is created in a window that opens up in your default browser, and there may be a fair delay after starting the program before the display appears.

The features of vpython are made available by importing everything in the vpython module with the statement:

```python
from vpython import *
```

and then invoking whatever statements are needed to create the desired display. The statements

```python
scene.autoscale = False  # Prevent scene from autoscaling.
scene.title = 'Projectile'  # Specify title in Scene window.
scene.background=color.white  # Set background to white.
scene.width=500  # Specify (in pixels) the width and height of the Scene window.
scene.height=500  # Set coordinates of center of scene.
scene.center=vector(10.0,0.0,0.0)  # Place text 'PROJECTILE' in scene, centered at (10.0, -4.0, 0.0).

vpython
```

set the characteristics of the scene.

We next define the objects in the scene—a sphere for the projectile and a box for the ground above which the ball moves—and stipulate that the moving ball is to leave a trail of its path with the statements:

```python
ball = sphere( pos=vector(0.0,0.0,0.0), radius=0.3, color=color.red, make_trail=True, trail_color=color.blue)
floor = box( pos=vector(10.0,0.0,0.0), size=vector(20.0,-0.3,0.0), color=color.green )
```

Here, the keyword `pos` specifies the location of the center of the object while the keyword `size` provides the $x,y,z$ dimensions of the box.

Then, we provide values for the constants and establish the initial velocity of the ball with the statements:

```python
g = 9.8; vx0 = 8.0; vy0=12.0
ball.velocity = vector(vx0,vy0,0.0)
```

---

105 The corresponding module for PYTHON 2 is the visual module, whose syntax is noticeably different from that of the vpython module. The visual module is being deprecated along with PYTHON 2, though it will remain available until 2020. Information about this module can be found at the URL vpython.org/contents/docs/VPythonIntro.pdf and an instructional PowerPoint presentation can be found at the URL www.cs.unc.edu/~lin/COMP089H/LEC/VPython.pptx. Apparently, the visual module works only with PYTHON 2.7.x installed from python.org. It will not work with PYTHON 2.7.x installed from other sources.

106 We import vpython using the construction "from vpython import *", allowing us to use components from the module without prefacing each with vpython. Had we used the construction import vpython as vp, say, the characters "vp." would have to precede any invocation of a component, e.g. vp.scene, vp.text, vp.color.red, ....

107 We are imagining a scene in which the horizontal coordinate runs from 0 at the left edge of the screen to 20 at the right edge. Thus, we position the center of the scene as here established and, later, set the center and width of a box to span from the horizontal coordinate from 0 to 20.

108 We here use the built-in objects sphere and box. The VPYTHON literature describes many other available objects.

109 Some trial and error preceded the selection of the values for the initial velocity.
Note that the second statement also adds a previously non-existent attribute to the object `ball`. Finally, we create the **while** loop

```python
t=0.0
deltat=0.01
while ball.pos.y > -0.001:
    rate(25)
    ball.pos = vector( ball.velocity.x*t, \n    -0.5*g*t**2+ball.velocity.y*t, 0.0 )
    t = t + deltat
```

controlled by the *y* position of the ball to update the velocity. In addition, the statement `rate(25)` controls the speed of the animation by specifying that a delay of at least \(1/25 = 0.04\) s occur before the next frame is displayed.\(^{110}\) In each cycle through the loop, the position of the ball is updated to the new time. With each update of the position by the program, the display on the screen is automatically updated. A complete listing of this program `projectile.py` is presented in Section 5.E.

Once `projectile.py` has been stored in an accessible directory, the program can be run in PYTHON 3 in any of at least three ways, specifically by

- opening the program in the IDLE **Edit** window and
  - selecting ‘Run Module’ from the RUN menu in that window or
  - making that window the active window and typing \(\text{F5}\) on the keyboard,
- setting the default directory in a command window to the operating system to the directory containing `projectile.py` and entering the statement `python projectile.py`, or
- setting the directory in the **Python Shell** to the directory containing `projectile.py` (See Section 5.16.1) and typing the command
  ```python
  exec( open('projectile.py').read() )
  ```
at the PYTHON 3 prompt.

We have here barely scratched the surface of the capabilities of VPYTHON. Numerous objects can be introduced into the scene.\(^{111}\) One can add buttons and sliders to the graphical display window. Further enumeration and illustration of these capabilities and many others is beyond the scope of this book.\(^{112}\)

### 5.15 Advanced Graphing Features

#### 5.15.1 Mathematical Symbols and Greek Characters

As illustrated in Fig. 5.5 and the associated PYTHON code, subscripts and italicized characters in text displayed on the screen can be produced by placing the characters to be so-represented in \LaTeXX math mode. e.g., \(\ldots\sin(\theta)\ldots\). \(^{113}\)

\(^{110}\) The delay will, of course, be longer if the processing of the loop requires more than 0.04 s.

\(^{111}\) See footnote 108.

\(^{112}\) More complete details can be found at any of several on-line references, including documentation and examples from links at the URL [www.vpython.org](http://www.vpython.org).

\(^{113}\) Alternatively duplicating backslashes in the character string can be avoided by preceding the string with the character \(r\), e.g., \(r'\ldots\sin(\theta)\ldots'\). This inclusion tells the PYTHON interpreter to pass the character string literally to the construction of the display.
5.15. ADVANCED GRAPHING FEATURES

Figure 5.26: Space curve of a charged particle moving in a constant magnetic field along the $z$ axis.

5.15.2 Space Curves

Sometimes it is desirable to view the trajectory of a particle in three-dimensional space. PYTHON has the ability of taking vectors of the $x$, $y$, and $z$ coordinates of points on a three-dimensional path, projecting these points onto the two-dimensional screen, and then connecting consecutive points with lines. To illustrate this feature, consider the equations

$$x = \cos t \quad ; \quad y = \sin t \quad ; \quad z = \alpha t$$ (5.19)

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 function plot3 (not to be confused with the plain vanilla function plot) to plot the graph. The statements

```python
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

t = np.linspace(0.0,30.0,201)
alpha = 0.5;
x = np.cos(t); y = np.sin(t); z = alpha*t

fig=plt.figure()
ax=plt.axes(projection='3d')
ax.set_xticks([-1.0,-0.5,0.0,0.5,1.0])
ax.set_yticks([-1.0,-0.5,0.0,0.5,1.0])
ax.set_zticks([0,5,10,15])
ax.plot(x,y,z,linewidth=2,color='black')
plt.show()
```

will produce the graph in Fig. 5.26.

5.15.3 Using Multiple Windows

PYTHON allows for the simultaneous presence of two or more separate windows on the screen. For example, the simple coding
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(0.0,10.0,101)
y1 = np.cos(x); y2 = np.sin(x)
fig1=plt.figure(1)
fig2=plt.figure(2)
ax1 = fig1.add_subplot(111)
ax1.plot(x,y1, linewidth=2, color='black')
ax1.set_title('$x$ versus $\cos(x)$', fontsize=14)
ax2 = fig2.add_subplot(111)
ax2.plot(x,y2, linewidth=2, color='black')
ax2.set_title('$x$ versus $\sin(x)$', fontsize=14)
plt.show()

will create two separate figures internally and then display them in separate windows in response to the statement `plt.show()`. Essentially, to display multiple windows, one needs to create all of them internally and then display them with a single invocation of `plt.show()`.

### 5.15.4 Scaling so Circles are Circles

Drawing circles that look like circles requires extra care in the specification of the axes, especially if the default coordinate ranges are different. Either with `plt.axis` or `ax.axis` (if the axis name `ax` has been defined), the relationship between the two axes in a 2D graph can be set, among others, to

- **auto**, which accepts the default aspect ratio for the plot window and scales each axis to embrace the data presented,
- **equal**, which accepts the default aspect ratio for the plot window but also sets the scaling on each axis so that equal coordinate increments correspond to equal lengths on the axes,
- **square**, which sets the aspect ratio so that the plot window is square and, if necessary, extends the range on each axis so that equal coordinate increments correspond to equal lengths on the axes, and
- **tight**, which accepts the default aspect ratio for the plot window and sets the limits on each axis so that there is little or no extra space around the graph.

To expose the differences among these options, we execute the statements shown in Table 5.7 to produce the display shown in Fig. 5.27. Notice that only the specifications **equal** and **square** create circles that appear as circles.

### 5.15.5 Customizing Axes

In PYTHON-generated graphs, a box around the graph will by default be generated, tick marks will be provided on each axis, and labels will be placed at the tick marks. For example, the simple unembellished coding listed on the left in Table 5.8 produces a 2D graph of the sine curve enclosed in a box with labeled tick marks along the left and bottom edges but no grid and the coding on the right produces a 3D graph of a displaced 2D membrane with grid, shaded planes on the left, back, and bottom sides of the graph and labels on all three coordinate axes.

The PYTHON module `matplotlib.pyplot` makes available several ways to modify the axes. The statement(s)

---

Additional options include off and on.
5.15. ADVANCED GRAPHING FEATURES

Table 5.7: Coding to illustrate making circles circles.

```python
import numpy as np
t = np.linspace(0.0,2.0*np.pi,101)
y = np.sin(t)
x1 = np.cos(t) + 1.25; x2 = x1 - 2.5

import matplotlib.pyplot as plt
fig, ( (ax1, ax2 ), ( ax3, ax4 )) = plt.subplots(2,2)
fig.subplots_adjust(hspace=0.35)
ax1.plot(x1,y,color='black')
ax1.plot(x2,y,color='black')
ax1.set_title('Axis auto',fontsize=14)
ax2.plot(x1,y,color='black')
ax2.plot(x2,y,color='black')
ax2.set_title('Axis equal',fontsize=14)
ax2.axis('equal')
ax3.plot(x1,y,color='black')
ax3.plot(x2,y,color='black')
ax3.set_title('Axis square',fontsize=14)
ax3.axis('square')
ax4.plot(x1,y,color='black')
ax4.plot(x2,y,color='black')
ax4.set_title('Axis tight',fontsize=14)
ax4.axis('tight')
plt.show()
```

Figure 5.27: The effect of different axis configurations.

![Axis auto](image1)
![Axis equal](image2)
![Axis square](image3)
![Axis tight](image4)
### CHAPTER 5. INTRODUCTION TO PYTHON

Table 5.8: Coding to produce 2D and 3D graphs with default axis positions.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
</table>
| # In 2D  
import numpy as np  
import matplotlib.pyplot as plt  
x = np.linspace(0.0,10.0,101)  
y = np.sin(x)  
plt.plot(x,y)  
plt.show() | # In 3D  
import numpy as np  
import matplotlib.pyplot as plt  
from mpl_toolkits.mplot3d import Axes3D  
xx = np.linspace(0.0,1.0,33)  
yy = np.linspace(0.0,1.0,33)  
x, y = np.meshgrid(xx, yy)  
z = np.sin(2.0*np.pi*x) * np.sin(3.0*np.pi*y)  
fig=plt.figure()  
ax = fig.add_subplot(111, projection='3d')  
ax.plot_surface(x, y, z, color='white', edgecolor='black')  
plt.show() |

- `plt.box('off')` added by itself in the 2D code above turns off the box around the graph but leaves the tick marks and their labels; added by itself in the 3D code, it has no effect.

- `plt.axis('off')` added by itself in the 2D code above turns off the box and the tick marks and labels, leaving the graph alone; added by itself in the 3D code above it turns off the tick marks, the labels, and the grided, shaded planes, leaving the graph alone.

- `plt.[x|y]ticks([List of Values])` added by themselves in the 2D code above will specify the coordinate values at which tick marks are to be placed. If the lists are both empty, tick marks and labels will be suppressed altogether. Whether the lists are empty or not, the box will remain.

- `ax.set_[x|y|z]ticks([List of Values])` added by themselves in the 3D code above will specify the coordinate values at which tick marks are to be placed and labeled. If the lists are all empty, tick marks and labels will be suppressed altogether. Whether the lists are empty or not, the shaded planes will remain.

As a more specific example, suppose we wanted to create a graph of \( y = \sin(x) \) over the interval \( 0 \leq x \leq 10 \) with the labeled \( y \)-axis drawn at \( x = 0 \) and the labeled \( x \)-axis drawn at \( y = 0 \). The coding, shown in Table 5.9, is a bit tedious because we have to take responsibility for almost every little detail. The resulting graph is shown in Fig. 5.28.

#### 5.15.6 Alternative Specifications of Color

In all previous examples, we have severely limited the colors used. Almost everywhere that the keyword `color` is used in commands in `matplotlib`, the color can be specified with

- one of `'blue', 'green', 'red', 'cyan', 'magenta', 'yellow', 'black', and 'white'`.  
- one of `'b', 'g', 'r', 'c', 'm', 'y', 'k', and 'w'`,  
- an RGB tuple, e.g., \((0.1, 0.3, 0.5)\), where the three values specify the amount of red, green and blue, respectively. All values must be in the range from zero to one. Here, \((0,0,0,0,0)\) corresponds to black, \((0.5,0.5,0.5)\) corresponds to a gray, and \((1.0,1.0,1.0)\) corresponds to white.  

---

115 The symbol | is the standard symbol for ‘or’. Thus, `[x|y]ticks` conveys either `xticks` or `yticks`.  

---

---
Table 5.9: Coding to produce graph of sin(x) vs. x.

```python
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(0.0, 10.0, 101)
y = np.sin(x)
plt.plot(x, y, color='black', linewidth=3)
plt.axis('off')
plt.axhline(y=0, color='black')
plt.axvline(x=0, color='black')
plt.hlines([-1.0, -0.5, 0.0, 0.5, 1.0], -0.3, 0.3, colors='black')
plt.vlines([0.0, 2.5, 5.0, 7.5, 10.0], -0.05, 0.05, color='black')
plt.text(10.5, -0.1, '$x$', fontsize=14)
plt.text(-1.5, 0.5, '$\sin(x)$', fontsize=14, rotation='vertical')
plt.text(-1.0, -1.02, '-1.0')
plt.text(-1.0, 0.98, '1.0')
plt.text(4.8, 0.1, '5.0')
plt.text(9.7, 0.1, '10.0')
plt.title('Graph of $\sin(x)$ vs. $x$', fontsize=16)
plt.show()
```

Figure 5.28: Graph of sin(x) vs. x.

- a hexadecimal RGB string, e.g., '#RRGGBB', where the three values RR, GG, and BB specify the amount of red, green, and blue, respectively. Each R, each G, and each B must be a hexadecimal digit between 0 and F. Here, '#FFFFFF' corresponds to white, '#0F0F0F' corresponds to a gray, and '#000000' corresponds to black. The specification is case-insensitive.

Many other specifications are supported.\(^{116}\)

\(^{116}\) See the URL matplotlib.org/users/colors.html for greater detail.
5.15.7 Interactive Plotting

In some contexts, you may not wish to wait until an entire graphical display has been constructed before seeing any of that display on the screen, i.e., you may wish to unblock the default PYTHON behavior that waits for the statement\texttt{plt.show()} before displaying anything. Many—though not all—PYTHON backends admit the statement \texttt{plt.ion()} to turn on \textit{interactive} mode.\footnote{We assume that the statement \texttt{import matplotlib.pyplot as plt} has been executed.} Once this statement has been executed with a background that supports interactive mode,\footnote{You can find out the backend in use in your environment by typing the statement \texttt{plt.get_backend()}.} the effect of each subsequent plotting statement is rendered immediately in the on-screen display. Thus, for example, if you execute the coding

```python
import matplotlib.pyplot as plt
import numpy as np
plt.ion()
x=np.linspace(0, 10, 101)
y=np.cos(x)
plt.plot(x,y)
plt.xlabel('x', fontsize=16)
plt.ylabel('cos(x)', fontsize=16)
```

each component of the display will be added to the display as the corresponding statement is executed. Note that the statement \texttt{plt.show()} has not been invoked. If you continue from the above with the statements

```python
plt.figure(2)
plt.plot(x,y,color='black', linewidth=3)
```

a second set of axes will be displayed on the screen and a blacker and thicker graph of the cosine function will appear.

Interactive plotting will be turned off by executing the statement \texttt{plt.ioff()}. The PYTHON prompt will return and additional statements can be submitted for execution. Any graphs created, however, will remain on the screen until they are manually removed.

This very brief discussion merely alerts you to a possibility. Many refinements, including the possibility of substituting an alternative backend for the default, are described in the PYTHON manuals or can be discovered by creative exploration.

5.16 Miscellaneous Occasionally Useful Tidbits

5.16.1 Find, Change Current Directory

Access from within PYTHON to a variety of operating system commands is provided by modules in the module \texttt{os}, which is imported with a statement like

```python
import os or from os import *
```

Once this module has been imported, the following commands—among many others—provide frequently used actions:

\footnote{You can determine if interactive mode is functioning by executing the two statements \texttt{import matplotlib; matplotlib.is_interactive()} after the statement \texttt{plt.ion()} has been executed.}

\footnote{Caution: Interactive mode will work only in the PYTHON command window. It is not available in the PYTHON Shell (IDLE).}
• `print(os.getcwd())`—report the current directory.

• `os.chdir('Path')`—change the current working directory to the directory identified by `Path`. In defining the path, forward slashes can be used to separate consecutive directories even if the native character for your operating system is not the forward slash.

• `os.system('OSCommand')` invokes the system command `OSCommand`.

• `os.listdir('Path')`—displays in the PYTHON window a list of all files in the directory identified by `Path`. Unfortunately, the list is hard to read because it sprawls across the entire screen; individual entries are not displayed in their own lines. If the module `numpy` has been imported, an easier-to-read listing would be created by the statement

\[
\text{print(numpy.array(os.listdir('Path')))}
\]

Many more features are provided in the `os` module.\textsuperscript{121}

### 5.16.2 Setting the Search Path

In searching for a specified file or module, PYTHON searches in the directories contained in its `search path`, which is set when PYTHON is launched by default or, perhaps, by statements in an initialization file—see Section 5.16.3—created by the user. To find out the search path or modify it, one must start by importing the `sys` module with the statement

\[
\text{import sys}
\]

Once this module is available, either of the statements `sys.path` or `print(sys.path)` will display a (more or less illegible) list of directories through which PYTHON searches to find a requested item. A more transparent listing is produced with either of the statements

\[
\text{for p in sys.path: print(p) or print('\n'.join(sys.path))}
\]

These statements, however, omit the first item in the illegible renditions of the search path. Specifically, the item `'\'`, which stands for the current directory, is always at the head of the path.\textsuperscript{122} Furthermore, because `sys.path` is a list, the \(n\)-th element in the list can be displayed with the simple statement `sys.path[n]`.

The `sys` module also contains components that will allow you to modify the search path. For example the statement

\[
\text{sys.path.append('Path')}
\]

will add the specified path at the end of the current path and the statement

\[
\text{sys.version or print(sys.version)}
\]

will display the version of PYTHON in use.

The module `sys` also contains the command `sys.getsizeof` for determining how much memory is used by a particular variable. The coding

\textsuperscript{121} See the URL docs.python.org/2/library/os.html.

\textsuperscript{122} If your—or your system manager on your behalf—has defined the environment variable PYTHONPATH, then the directories in that path will follow the directory `'\' and precede all the specifically PYTHON directories that are included in the search path.
a = [1,2,3,4,5,6,7,8,9,10]
sys.getsizeof( a )
  144
b='David M. Cook'
sys.getsizeof( b )
  46

reveals that the list a requires 144 bytes of storage while the character string b requires 46 bytes of storage.

5.16.3 Customizing PYTHON

If you find yourself frequently beginning your sessions with PYTHON by executing the same few statements, maybe importing a few modules, you may find it convenient to create a command file with those several statements, store it in an accessible directory, and define the environment variable PYTHONSTARTUP to point to that file. For example, if you are working in a Windows environment

- Create the file pythonstart.py containing the lines

  ```python
  import numpy as np
  import matplotlib.pyplot as plt
  import os
  os.chdir( 'c:/users/myhome/documents/examples/python' )
  ```

  or whatever directory is appropriate to your task.

- Store this file in an accessible directory, e.g., `c:\users\cookd\python` and

- In the command window in which you are working, define the environment variable PYTHONSTARTUP with the statement

  ```
  set PYTHONSTARTUP=C:\users\cookd\python\pythonstart.py
  ```

the startup file will be found and executed if either PYTHON 2 or PYTHON 3 is launched with the command python from the same command window in which the set command was issued but will not be recognized in any subsequently created command window.

If, instead, you are working in a UNIX environment in a C-shell or T-shell and the startup file pythonstart.py is stored, for example, in the directory `/home/cookd/python`, the statement

```bash
setenv PYTHONSTARTUP /home/cookd/python/pythonstart.py
```

will establish that environment variable for use in the current command window. Subsequently, the startup file will be found and executed if either PYTHON 2 or PYTHON 3 is launched with the

---

123 The simple statement `set` will display a list of all environment variables; the statement `echo %PYTHONSTARTUP%` will display only the specified environment variable.

124 If you wish to set this environment variable so that it will be automatically created when you log in in any command window, you will need the statement `setx PYTHONSTARTUP C:\users\cookd\python\pythonstart.py`, though you will have to launch a new command window for this change to take effect.

125 The commands are slightly different in other shells. See the manual for the shell you are using or consult with a system administrator.

126 The simple statement `setenv` will display a list of all environment variables; the statement `echo $PYTHONSTARTUP` will display only the specified environment variable.
command `python` from the same command window in which the `setenv` command was issued but will not be recognized in any subsequently created command window.\footnote{If you wish to set this environment variable so that it will be automatically created when you log in in any command window, you place the command setting the environment variable in the `.cshrc` file in your home directory.}

In none of the above cases in Windows or UNIX will the startup file be found if PYTHON 2 or PYTHON 3 is launched by issuing the command `idle` at a command prompt or by clicking ML on the IDLE (Python GUI) icon, i.e., if you bring up the Python Shell in any way. Further, routes to arrange for that file to be executed automatically when the Python Shell is launched appear to be cumbersome. The easiest route to customize the Python Shell is to execute that startup file from the PYTHON prompt using the command `execfile` (PYTHON 2) or `exec` (PYTHON 3).\footnote{See Sections 5.6.1, 5.7.2, 5.7.3, 5.14.2, 5.14.3, and 5.14.4.}

Note also that, through the item ‘Configure IDLE’ in the the OPTIONS menu in the Python Shell, you can adjust the behavior of that shell. In the several tabs in that menu, one can select the font, the font size, bold font, the size of PYTHON required indents, the colors for highlighting various components of the on-screen text, and many other behaviors. In particular, in the GENERAL tab, you can, among other options, (1) specify whether the Edit window or the Shell window is to be launched when IDLE is launched and (2) specify whether when you run a file from the Edit window you should be prompted to save the file if it has not been saved.

The module `matplotlib` can also be customized in a variety of ways. If, for example, you tire of repeatedly typing the keywords `linewidth` and `fontsize`, you can customize these features of all subsequent graphs by executing the statements\footnote{With `mpl` defined as here, the statement `mpl.rcParams` will display a list of all–almost 300—of the adjustable parameters in the dictionary-like variable `mpl.rcParams` along with the current value of each. The statement `mpl.rcdefaults()` will restore all default parameters.}:

```python
import matplotlib as mpl
from cycler import cycler
mpl.rcParams['lines.linewidth']=4
mpl.rcParams['xtick.labelsize']=16
mpl.rcParams['ytick.labelsize']=16
mpl.rcParams['axes.titlesize']=20
mpl.rcParams['axes.prop_cycle']=cycler('color', ['r'])
```

Most easily, statements can be

- entered individually and interactively at a PYTHON prompt, either in the CLI or in the Python Shell, or
- incorporated in a file of your chosen name, stored in an accessible directory, and executed at a PYTHON prompt with the command `execfile` (PYTHON 2) or `exec` (PYTHON 3),\footnote{Color adjustments is complicated and requires the function `cycler` from the module `cycler`. Adjusting `axes.prop_cycle` can specify a single color or the cycle of colors through which overlayed graphs will pass.}

Alternatively (though this route is more complicated), you can copy the file `matplotlibrc` from the directory in which PYTHON itself is stored,\footnote{See footnote 128.} edit that file to reflect the desired adjustments, and store it in a file named `matplotlibrc` in

- the directory from which you will launch PYTHON in a command window to the operating system,
- the directory pointed to by the environment variable `MATPLOTLIBRC` (which must, of course, be defined), or

\footnote{You can find the location of the standard `matplotlibrc` file by importing `matplotlib` as `mpl` and then executing the statement `print( mpl.matplotlib_fname() )`.}
• a user-specific subdirectory (usually .config/matplotlib in the UNIX world and .matplotlib on other platforms) in the user’s home directory.

In this case the configuration file will be found automatically but only if PYTHON is launched in a command window to the operating system; it will not be found if the Python Shell is launched by clicking on an icon or by a command to the operating system.\textsuperscript{133}

### 5.16.4 Restoring PYTHON’s Initial State

The simplest way to restore PYTHON’s initial state is to exit from and then re-launch PYTHON, and this approach will always work, however you started PYTHON in the first instance. Of course, with a restart, PYTHON is returned to its fresh-start state; nothing from your previous activity survives the restart.

If you are creating code in the PYTHON Edit window, PYTHON will be automatically restarted every time you select ‘Run Module’ from the Run menu in the Edit window or, alternatively, whenever you type (F5) when the Edit window is active. Again, nothing from the previous execution of your code survives the restart, and everything needed must be explicit in the code on which you are working.

Suppose, on the other hand, you are working directly and interactively in the Python Shell. Instead of exiting and re-launching PYTHON, you can select ‘Restart Shell’ from the Shell menu along the top of the window (or type (CONTROL-F6)). Yet again, nothing from your previous activity survives the restart.

Finally, if you execute a PYTHON program directly from the prompt in a shell of the operating system with a statement like

```
python standingwave.py
```

control will be returned automatically to the operating system when the program is completed, and PYTHON will be started afresh with each new command to the operating system.

### 5.16.5 Reading Data from the Keyboard

The simple command `input` in a PYTHON program requests input from the keyboard at execution time. In PYTHON 2, the general structure of a statement utilizing this command is

```
Variable(s) = input( 'Prompt: ' )
```

The format of the input is dictated by the nature of the specified variables. For example, the statement

```
a = input( 'Enter a value: ' )
```

will expect a single quantity to be entered, though that quantity may be a string, a numeric value, a tuple, a list, a logical value, a logical condition, and perhaps many other entities. In PYTHON 2, the data type of the variable `a` will be fixed dynamically, depending on the data type of the entered quantity. Further, in PYTHON 2, the statement

```
a,b = input( 'Enter two values, separated by a comma: ' )
```

\textsuperscript{133} See the URL matplotlib.org/users/customizing.html for greater detail on the topic of this paragraph.
5.17. REFERENCES

will request input of two values, each of which can have any available data type.

In PYTHON 3, only a single variable can be returned and the general structure of a statement utilizing this command is

\[ \text{Variable} = \text{input( 'Prompt: ' )} \]

In PYTHON 3, only one variable can be input and the value input, even if it has several components separated by commas or spaces, will be stored initially as a single string which must be parsed and converted if necessary.\(^{134}\) In any given context, it is prudent to include in the prompt a stipulation of the data expected as input.

5.16.6 Error Trapping

In order to avoid unanticipated problems, it may sometimes be wise to trap invalid entries and give the user an opportunity to enter a valid entry before moving on to the remainder of the program. For example, in PYTHON 2, if an integer is needed, the coding

\[
a = 'string' \\
\text{while isn} \	ext{stance( a, int )} == \text{False:} \\
\quad a = \text{input( 'Please enter an integer: ' )}
\]

will ask for an integer and ask repeatedly until an integer is provided. The loop is primed by setting a equal to a non-integral value.

The coding just presented works in PYTHON 2 because the data type of the value entered for \(a\) is dynamically set. In PYTHON 3, the output from an \texttt{input} statement is always typed as a string, so the loop above will run forever and never terminate, even if something that looks like an integer is entered.\(^{135}\) A better route in PYTHON 3 involves exploiting the \texttt{try/except} control structure. For example, the coding\(^{136}\)

\[
a = \text{None} \\
\text{while a == None:} \\
\quad \text{try:} \\
\quad \quad a = \text{int(input("Please enter an integer: "))} \\
\quad \quad \text{except ValueError:} \\
\quad \quad \quad \text{print("Error: You must enter an integer!")}
\]

creates a loop that asks for an integer, tests if an integer is actually entered, and smoothly asks for another entry if the first entry fails that test (and displays only the user-supplied messages). Once the test is satisfied, the loop terminates with the proper value assigned to the variable \(a\). This coding will also work in PYTHON 2, though entry of a floating value will truncate that value to the highest integer not exceeding that value and succeed.

5.17 References

Beyond the links enumerated in Section 5.8, numerous resources are available to supplement those identified in Section 5.8. Searching for the specific item of interest in your favorite web browser will

\(^{134}\) See the PYTHON attribute \texttt{split} encountered first in Section 5.6.2.
\(^{135}\) Entering 3 will set \(a\) to the string ‘3’.
\(^{136}\) \texttt{ValueError} is one of the types of error that can be tested. Others include \texttt{IOError, ImportError, EOFError}, and \texttt{KeyboardInterrupt}. Using \texttt{except} without any error type will trap all errors.
likely generate useful links. For example, searching in your browser for 'lists in PYTHON', 'array in numpy', 'fontsize in PYTHON', and 'subplots in matplotlib' will generate several hundred thousand hits. Going to www.bn.com or www.amazon.com and searching for book titles containing 'PYTHON' or 'NUMPY' or 'MATPLOTLIB' will yield an overwhelming number of hits, the first couple dozen or so of which may well be worth considering. The websites

- www.python.org—the PYTHON home site, which provides full information about PYTHON and, in particular, has a link to a huge volume of official PYTHON documentation.
- docs.python.org/?/tutorial/index.html, where ? = 2 or 3, points you to the official PYTHON tutorial.
- www.numpy.org—the numpy home site, which provides full information about numpy.
- docs.scipy.org/doc/, which contains links to many documents describing numpy and its features.
- matplotlib.org/—the matplotlib home site, which provides full information about matplotlib and, in particular, has links to tutorials and documents in a band near the top of the page.
- www.tutorialspoint.com/python/python_basic_syntax.htm, which links to a third-party site with a variety of elementary tutorials focussed, unfortunately, on an older version of PYTHON.

Searching your desired information on the web may well be more efficient than trying to track it down in the official manuals provided by the suppliers of PYTHON and its modules.

Several links to URLs providing information on more narrow topics have been included in footnotes 14, 21, 31, 35, 66, 80, 85, 93, 94, 108, 112, 116, 121, and 133, in Table 5.6, and at the end of Sections 5.14.1, 5.14.4, and 5.16.1.

5.18 Exercises

5.18.1 Writing PYTHON Statements

5.1. Write and test PYTHON statements to create (a) a five-element column array, (b) an $8 \times 8$ unit matrix, and (c) a $10 \times 10$ 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 $10 \times 10$ matrix individually.

5.2. Look up the attribute sort and the command sorted in the PYTHON manuals. Then, create a list of your choice and test the use of sort and sorted, following the pattern illustrated in the documentation. Finally, write in your own words a brief description of what sort and sorted do, and make clear the distinction between the two.

5.3. The scipy.linalg module, which may or may not be installed in your version of PYTHON, includes the command scipy.linalg.lu that will effect a lower-upper (LU) decomposition of a square matrix, i.e, recast a single matrix $A$ as the product $LU$, where $L$ is a matrix all of whose elements above the main diagonal are zero and $U$ is a matrix all of whose elements below the main diagonal are zero. More specifically, the equation $Ax = b$ becomes

\[
LUx = b \implies \begin{pmatrix}
1 & 0 & 0 \\
L_{21} & 1 & 0 \\
L_{31} & L_{32} & 1
\end{pmatrix}
\begin{pmatrix}
U_{11} & U_{12} & U_{13} \\
0 & U_{22} & U_{23} \\
0 & 0 & U_{33}
\end{pmatrix}
\begin{pmatrix}
x_1 \\
x_2 \\
x_3
\end{pmatrix} =
\begin{pmatrix}
b_1 \\
b_2 \\
b_3
\end{pmatrix}
\] (5.20)
Once we have the LU decomposition in hand, we (1) rename $Ux$ to $y$, (2) solve $Ly = b$ for $y$ (which is easy by backsubstitution, since the first row of $Ly = b$ tells us $y_1 = b_1$, the second row then tells us that $L_{21}y_1 + y_2 = b_2$, ...), and then we solve $Ux = y$ by a similar approach. Look up the commands `scipy.linalg.lu_factor` and `scipy.linalg.lu_solve` in the PYTHON manuals, and then use the process described therein to solve the linear equation solved by other means in Section 5.3.7.

5.4. Describe and test a sequence of PYTHON statements that uses a `for` loop to evaluate the dot product of two one-dimensional $n$-component arrays $a$ 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.

5.5. (a) Describe and test a sequence of PYTHON statements that uses a `for` loop to evaluate $\sum_i a_i$ when the values of $a_i$ are supplied as the elements of a list $a$. 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 PYTHON statements that uses the function `numpy.sum` to achieve the same end. (c) Test whether the procedure used in (b) for a list also works for a one dimensional array. (c) Explore and then describe the behavior of the function `numpy.sum` when it is applied to a list of lists. Explain the difference between the statements `numpy.sum(a)`, `numpy.sum(a, axis=0)`, and `numpy.sum(a, axis=1)` when applied to an array $a$ containing three rows and four columns.

5.6. Write and test a PYTHON function that accepts two one-dimensional three-component lists or arrays as input and returns a one-dimensional three-component list or array containing the cross product of the two input arguments. Verify that your function properly reflect as the fact that $A \times B = -B \times A$.

5.7. 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.

### 5.18.2 Finding Eigenvalues and Eigenvectors

5.8. Using `numpy.linalg.eig` for the main calculation, write and test a function that accepts a symmetric matrix as input and returns a single matrix, each column of which contains an eigenvalue as its first element and the associated eigenvector as its remaining elements.

5.9. 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 $10 \times 10$ matrix with zeroes everywhere except that all elements on the main diagonal have the value 2.0 and all elements on the sub and superdiagonals have the value $-1.0$. To create the matrix, design a method more efficient than laboriously setting each of the 100 elements in the $10 \times 10$ matrix individually.

(c) Identify the five eigenvectors 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. Actually, in the underlying physical context, it would be more appropriate to plot graphs of the result of augmenting these eigenvectors
CHAPTER 5. INTRODUCTION TO PYTHON

5.10. 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 energies of the states with principal quantum number $n$ shift from the energies 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 \([2,0,0], [2,1,-1], [2,1,0], \) and \([2,1,1] \), then the matrix for the state $n = 2$ is

\[
3ea_0F \begin{pmatrix}
0 & 0 & -1 & 0 \\
0 & 0 & 0 & 0 \\
-1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0
\end{pmatrix}
\]

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 \([3,2,2], [3,1,1], [3,2,1], [3,0,0], [3,1,0], [3,2,0], [3,1,-1], [3,2,-1], \) and \([3,2,-2] \), then the matrix for the state $n = 3$ is

\[
3ea_0F \begin{pmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & -9/2 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & -3\sqrt{6} & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -3\sqrt{6} & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & -9/\sqrt{3} & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & -9/\sqrt{3} & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & -9/2 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -9/2 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}
\]

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.

5.18.3 Graphing Scalar Functions of a Single Variable

5.11. Create and test a sequence of PYTHON statements that will produce a graph that is similar to Fig. 5.6 except that the axes are drawn along the lines $y = 0$ and $x = 0$.

5.12. (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.
5.13. 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 \( z/a \).

5.14. 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.

5.15. 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 \( \delta \) for various values of \( r \), and write a paragraph describing these graphs.

5.16. 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.

5.17. 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

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

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.

5.18. 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

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

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.

5.19. 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

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

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

5.21. 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 a PYTHON procedure to produce such a graph and, if you have suitable data available (or if you can invent some), demonstrate that your procedure works.

5.22. When a photon of initial energy $E_0$ undergoes Compton scattering with 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.

5.23. 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 $\beta = v/c$. Obtain graphs of $E/(q/4\pi \epsilon_0 r^2)$ versus $\theta$ for various values of $\beta$ on the interval $-\pi \leq \theta \leq \pi$, and write a paragraph describing these graphs.

5.24. 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.

5.18.4 Graphing Scalar Functions of Two Variables

5.25. 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 $\tau = T/T_0$. (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

$$u(\lambda, T) = \frac{1/\Lambda^5}{e^{ch/(\Lambda k \tau)} - 1}$$

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

5.26. 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 $B(z)/B_0$ 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.

5.27. 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

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

where $B_0$ is the magnetic field at the origin. Plot graphs showing $B(z)/B_0$ 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.

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

$$I = \frac{I_0}{\sqrt{1 + Q^2 \left( \frac{\Omega}{\Omega - \frac{1}{\Omega}} \right)^2}}$$

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.

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

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

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 $175 \, ^\circ C$, i.e., to a value such that $u(0, t)/u_0 = 0.875$. 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 $[\kappa = 0.63 \, J/(m s K)]$ and $c = 4.2 \times 10^6 \, J/(K m^3)]$, 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.

5.30. 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 PYTHON, 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 $z = 0.5R$. **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 $GmM/R$.

### 5.18.5 Graphing Scalar Functions of Three Variables

5.31. Following the pattern illustrated in Section 5.12, explore at least one of the three-dimensional scalar fields

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

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

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

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

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.

5.32. 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 $t = 0$ and at $t = \pi/\omega_{lmn}$ for several different values of $l$, $m$, and $n$.

### 5.18.6 Graphing Vector Fields

5.33. 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

\[
x0 \leftarrow x_0 \quad \text{Set starting point.}
\]

\[
y0 \leftarrow y_0 \quad \text{Choose step.}
\]

\[
ds \leftarrow \text{chosen step size}
\]

\[
\text{LOOP}
\]

\[
FX \leftarrow F_x(x0, y0) \quad \text{Start loop to generate steps.}
\]

\[
FY \leftarrow F_y(x0, y0) \quad \text{Calculate components of force.}
\]

\[
FM \leftarrow \text{SQRT}(FX^2 + FY^2) \quad \text{Calculate magnitude of force.}
\]

\[
xnew \leftarrow x0 + ds*FX/FM \quad \text{Step $dx$ along force line.}
\]

\[
ynew \leftarrow y0 + ds*FY/FM \quad \text{Step $dy$ along force line.}
\]

\[
\text{Draw line from $(x0, y0)$ to $(xnew, ynew)$}
\]

\[
x0 \leftarrow xnew \quad \text{Draw segment of force line.}
\]

\[
y0 \leftarrow ynew \quad \text{Shift segment to new point.}
\]

\[
\text{EXIT LOOP WHEN DONE}
\]

\[
\text{END LOOP}
\]

\[
\text{Repeat.}
\]

Cast the loop in this algorithm as a PYTHON procedure `trace_field` that would be called with a statement like

\[
\text{trace_field( x0, y0, ds, n )}
\]
where \( x_{\text{old}} \) and \( y_{\text{old}} \) convey the desired starting point for a field line, \( ds \) conveys the desired step size, and \( n \) 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, \texttt{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. \textit{Hint:} You may want to use the command \texttt{plot} with the keyword \texttt{nodata} set to \texttt{true} to establish the axes before beginning to trace any field lines. \textit{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 \( x_{\text{new}} \) and \( y_{\text{new}} \) determined above are instead regarded as \( x_{\text{pred}} \) and \( y_{\text{pred}} \), the field is calculated at that predicted point, and then a final step to \( x_{\text{new}} \) and \( y_{\text{new}} \) is made from \( x_{\text{old}} \) and \( y_{\text{old}} \) by using the average of the fields at the points \((x_{\text{old}}, y_{\text{old}})\) and \((x_{\text{pred}}, y_{\text{pred}})\).

5.34. 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 \( x_{\text{new}} \) and \( y_{\text{new}} \) in the algorithm in the previous exercise with the statements

\[
x_{\text{new}} \leftarrow x_{\text{old}} - ds \cdot FY/FM
\]

\[
y_{\text{new}} \leftarrow y_{\text{old}} + ds \cdot FX/FM
\]

Following a pattern similar to that described in the previous exercise, cast the loop in this algorithm as a PYTHON procedure \texttt{trace_equipot} that would be called with a statement like

\texttt{trace_equipot( xold, yold, ds, n )}

where \( x_{\text{old}} \) and \( y_{\text{old}} \) 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.

5.35. Sometimes, interest can be focussed on the \textit{magnitude} of the vector field rather than on the full field. A command like

\[
\text{MAG} = \text{numpy.sqrt( Hx**2 + Hy**2 )}
\]
for example, will generate an array \( \text{MAG} \) 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 5.11. Explore the magnetic field given in Eq. (5.14) in this way.

5.36. Patterning your solution after the discussion in Section 5.13.3, use the PYTHON attribute `quiver` to explore the velocity field

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

(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. Hint: In establishing a scale, you may find it useful to assume a length scale \( a \), recast the field in the form

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

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

5.18.7 Animation

5.37. 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

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

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 \( 1/n \).

5.38. 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 py-file to accomplish this task. Try your procedure with the two functions

\[
f(x, t) = e^{-(x-vt)^2} \quad \text{and} \quad f(x, t) = \sin(x) \cos(t)
\]

but feel free to invent others of your own choosing.

5.39. 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 function M-file to animate the motion of this membrane 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 5.14.
5.A Listing of plotradio.py

>>> Program plotradio.py

This PYTHON command file reads the file radio.dat and creates a graph showing the time variation of three species as the first decays to the second and the second decays to the third.

>>> # Import necessary packages.

import matplotlib.pyplot as plt
import numpy as np

# Open and read the file; convert string input to numbers and then convert list to an array.

f = open( "radio.dat", 'r' )
ln=f.readline()
data = []
for line in f:
    data.append( [float(x) for x in line.split()] )
f.close()
dataarray = np.array(data)

# Assign a variable name to each column in the array.

t = dataarray[:,0]; A = dataarray[:,1]
B = dataarray[:,2]; C = dataarray[:,3]

# Plot the graph.

plt.plot( t, A, color='black', linewidth=3 )
plt.plot( t, B, color='black', linewidth=3 )
plt.plot( t, C, color='black', linewidth=3 )
plt.grid( color='black' )
plt.text( 2.5, 850.0, '$A$', fontsize=16 )
plt.text( 2.5, 300.0, '$B$', fontsize=16 )
plt.text(40.0, 800.0, '$C$', fontsize=16 )
plt.show()

5.B Listing of rcfilter.py

>>> Program rcfilter.py

This PYTHON command file reads the file rcdata.dat and creates a graph showing the gain of an RC filter versus the frequency of the signal presented at the filter's input. Error bars and the theoretical expectation are also shown.
# Import necessary packages.
import matplotlib.pyplot as plt
import numpy as np

# Read data file and calculate uncertainties at 5%
data = np.loadtxt('rcdata.dat')
unc = data[:,1]*0.05

# Plot the semilog graph, add error bars and labels.
plt.semilogx( data[:,0], data[:,1], 'o',
              markerfacecolor='none', markeredgecolor='black')
plt.errorbar( data[:,0], data[:,1], unc, 
              ecolor='black', linestyle='none',
              capsize=2 )
plt.title('Experimental Data for High Pass Filter', fontsize=20)
plt.xlabel('Frequency (Hz)', fontsize=14)
plt.ylabel('Gain (Vout/Vin)', fontsize=14)

# Add theoretical curve.
r = 1.5e4
c = 0.0442e-6
f0 = 1/(2.0*np.pi*r*c)
temp = np.arange( 1.0, 4.1, 0.12 )
f = np.exp(2.3026 * temp)
gtheory = 1.0/np.sqrt(1.0 + f0**2./f**2)
plt.plot( f, gtheory, 'k--', linewidth=2 )
plt.plot( [500.0, 900.0], [0.325, 0.325], 'k--' )
plt.text( 1000.0, 0.30, 'Theoretical', fontsize=16 )
plt.plot( [700.0], [0.23], 'o', markerfacecolor='none',
          markeredgecolor='black' )
plt.text( 1000.0, 0.20, 'Experimental', fontsize=16 )
plt.grid( color='black' )
plt.show()

5.C  Listing of stwave.py

Program stwave.py

This program generates an animated display of a standing wave
in a string.

# Import necessary packages
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import animation

# Create an empty figure with fixed axes, add a grid, and create an
# empty variable line to store the coordinates of the points on the
# line to be drawn in each frame as it is generated.
fig = plt.figure()
ax = plt.axes(xlim=(0,1), ylim=(-1.5,1.5))
ax.grid()
line, = ax.plot([],[],linewidth=2, color='black')

# Define functions to initialize the variable line and create the
# several frames in the display.
def init():
    line.set_data([],[])
    return line,
def animate(i, harm):
    x = np.linspace(0.0,1.0,101)
y = np.cos(i*np.pi/50.0)*np.sin(harm*np.pi*x)
    line.set_data(x,y)
    return line,

# Produce the frames in the animation, storing them internally.
harm = int( input('Specify harmonic: ' ) )
anim = animation.FuncAnimation( fig, animate,init_func=init, 
    frames=100, interval=20, repeat=True, fargs=(harm,) )

# Display the animation on the screen.
plt.show()

5.D  Listing of drumhead.py

##
### Program drumhead.py

This program generates an animated display of the motion of
an L-shaped drumhead.

##
def surf(k):
    ax.clear()
    ax.set_zlim( (-0.25,0.25) )
    pic = ax.plot_surface( x,y,drm[k], color='white', edgecolor='black' )
return pic,

# Import necessary packages.

import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import animation

# Extract all of the data from the file.

f = open( 'ldrum.dat', 'r' )
data = []
for line in f:
    data.append( [ float(x) for x in line.split() ] )

f.close()

# Convert data into 41 33x33 arrays containing the shape
# of the drumhead at each time.

i = 0
k = 0
drm=np.zeros( (41,33,33) )
while k <= 40:
    tmp = []
    j = 0
    while j <= 32:
        nxtln = np.concatenate([data[i],data[i+1],data[i+2],
                                data[i+3],data[i+4],data[i+5]], axis=0)
        tmp =np.concatenate( [tmp, nxtln], axis=0 )
        i=i+6
        j=j+1
    drm[k] = np.reshape(tmp, (33,33) )
    k = k+1

# Set up the axes for the display.

xx=np.linspace(0.0,1.0,33); yy = np.linspace(0.0,1.0,33)
x, y = np.meshgrid( xx, yy )
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.set_xlim( (0.0,1.0) )
ax.set_ylim( (0.0,1.0) )
ax.set_zlim( (-0.25,0.25) )

# Create all frames and display them.

anim = animation.FuncAnimation( fig, surf, frames=41, repeat=True )
plt.show()
5.E Listing of projectile.py

```python
""" projectile.py

Plots the trajectory of a projectile launched at an angle to the horizontal as it moves near the earth under gravity and without air resistance.
"""

from vpython import *  # Import vpython module

# ***** Set particulars of the scene *****

scene.autoscale = False
scene.title = 'Projectile'
scene.background=color.white
scene.width=500
scene.height=500
scene.center=vector(10.0,0.0,0.0)
text(text='PROJECTILE', align='center',
     pos=vector(10.0,-4.0,0.0),
     color=color.black, height=1.0, font='sans')

# ***** Create projectile and floor *****

ball = sphere( pos=vector(0.0,0.0,0.0), radius=0.3, color=color.red,
               make_trail=True, trail_color=color.blue)
floor = box( pos=vector(10.0,0.0,0.0), size=vector(20.0,-0.3,0.0),
            color=color.green)

# ***** Set initial values *****

g = 9.8; vx0 = 8.0; vy0=12.0
ball.velocity = vector(vx0,vy0,0.0)
t=0.0
deltat=0.01

# ***** Repeatedly update position of projectile, stopping
# y coordinates first becomes negative *****

while ball.pos.y > -0.001:
    rate(25)
    ball.pos = vector( ball.velocity.x*t, 
                      -0.5*g*t**2+ball.velocity.y*t, 0.0 )
    t = t + deltat
```
Chapter 8

Introduction to MATHEMATICA

Note: All program (.m) files referred to in this chapter are available in the directory $HEAD/mathematica, 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 Mathematica’s default search path. If so, the files can be found by Mathematica 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.

Conceived and developed by Stephen Wolfram and first released to the public in June of 1988, Mathematica® is a large program for manipulating expressions symbolically.1 Mathematica 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 Mathematica and then illustrate some of the statements by which Mathematica can be instructed to manipulate these expressions. Users should also work through the on-line help messages within Mathematica itself and, in particular, look at the examples with which most on-line help messages conclude (Section 8.2). Further details can be found in printed and on-line Mathematica documentation (Section 8.19). We shall refer to this documentation collectively as the Mathematica manuals.

Two user interfaces to Mathematica are provided, a text-based interface (TBI), which gives direct access to the Mathematica kernel, and a notebook interface (NBI), 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. In essence, the TBI gives the user direct access to the Mathematica kernel while the NBI is simply a front-end that facilitates construction of statements that the NBI then formulates properly and passes on to the kernel, which may actually be running on a different (and perhaps faster) computer than is used for the front-end. In this book, we assume that the user is working in the NBI. We will, however, describe the structure of input statements without addressing the ways in which the NBI facilitates assembling them,2 so the statements we describe can equally well be submitted to the TBI. We shall present Mathematica output with the approximate appearance it will have in the NBI.

Except where occasionally noted otherwise, for the rest of this book, information provided describes the NBI explicitly and may not accurately reflect the behavior of the TBI.

---

1 Mathematica is a registered trademark of and also a commercially available program produced and marketed by Wolfram Research, Inc., of Champaign, Illinois. (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 Wolfram Research, Inc. The terms of that license are explained in the Local Guide.

2 See the Mathematica manuals for information about how to use templates, palettes, and other aids available in the NBI.
8.1 Beginning a *Mathematica* Session

Detailed instructions for initiating a session with Mathematica will be found in the *Local Guide*. In general, Mathematica will be started either (1) by typing the command `mathematica` for the NBI (or `math` for the TBI) at the prompt from the operating system,\(^3\) (2) by double-clicking ML on an appropriate icon on the desktop, or (3) by selecting ‘Mathematica’ (or an equivalent phrase) from a menu.\(^4\) Presently, an introductory screen offering a number of options will appear. Opening the New Document drop-down menu and selecting the option ‘Notebook’ produces a blank notebook titled ‘Untitled-1’ and a cursor—but no prompting character(s). We must from the outset be aware that

1. Labels, e.g., ‘In[12]:=’ for input and ‘Out[12]=’ for output, will appear after a statement has been submitted to Mathematica.

2. Internally, Mathematica is case sensitive, i.e., Mathematica treats upper and lower case letters as distinct. By convention, built-in Mathematica commands use names whose *initial* letter is capitalized.

3. By default, Mathematica formats input statements using Wolfram Language input. Other formats can be selected by opening the menu accessed by the ‘+’ sign that appears at the top of the notebook.

4. Lines of Mathematica code can be terminated in any of three ways, depending on whether we want to

   (a) submit the statement for execution and instruct Mathematica to display its response, in which case we terminate the statement by typing ⟨SHIFT/ENTER⟩ on the main keyboard or by typing ⟨ENTER⟩ on the numeric keypad.\(^5\)

   (b) submit the statement for execution and instruct Mathematica to suppress explicit display of its response, in which case we terminate the statement with a semicolon followed by ⟨SHIFT/ENTER⟩ on the main keyboard or ⟨ENTER⟩ on the numeric keypad.

   (c) continue entering the statement on a new line, in which case we terminate the line by typing simply ⟨ENTER⟩, but we must be clever in exploiting this feature. Mathematica will expect additional input *only* if the part entered before typing ⟨ENTER⟩ is incomplete. To enter the statement \(a + b + c\) on two lines, for example, we could not break it after the \(b\) but we could break it after either of the + signs.

5. Two or more statements can be placed in a single physical line by separating the statements with semicolons, though output will be suppressed for all but the last statement in the line (unless it is followed by a semicolon).

6. To abort execution of a statement by the Mathematica kernel and return Mathematica to a state from which new statements can be entered, either type ⟨ALT/⟩ or, alternatively,
   
   - Select the block containing the statement whose execution is to be terminated by clicking ML on the corresponding bracket at the right end of the screen,
   - Open the Evaluation menu from the menu bar across the top of the notebook, and
   - Select ‘Abort Evaluation’ from that menu.

   There may be a short delay during which Mathematica will clean up its workspace before actually “prompting” for fresh input.

---

\(^3\)In this command to the operating system, case may be important.
\(^4\)Access to Mathematica is controlled by a license manager. When Mathematica is started, the license manager checks to make sure 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.
\(^5\)On some keyboards the ⟨ENTER⟩ key will be labeled ⟨RETURN⟩.
7. Typing the statement Quit[] whenever *Mathematica* is ready for input cleans *Mathematica*’s workspace and reinitializes its internal statement counter but does not close the notebook or return control to the operating system. To exit altogether from *Mathematica*, select ‘Exit’ from the File menu or click ML on the ‘×’ button in the upper right corner above the *Mathematica* menu bar. If the notebook contains unsaved content, a pop-up box will give you the option of saving that content before *Mathematica* actually returns control to the computer’s operating system.

8. To close the current notebook without exiting from *Mathematica*, select ‘Close’ from the File menu. If, after closing a notebook, you open a new notebook by selecting ‘New→Notebook’ from the File menu or ‘Notebook’ from the *Mathematica* opening screen, statement numbers in that new notebook will continue from where they left off in the closed notebook. To reinitialize the internal statement counter, execute the statement Quit[], preferably before closing the first notebook but alternatively right after opening the new notebook.

9. The *Mathematica* command Run[ "OSCommand" ] effects temporary escape from *Mathematica* to the underlying operating system (OS) to effect execution of OSCommand by that operating system. Control is then returned to *Mathematica*. The resulting output (if any) is displayed in the command window from which *Mathematica* was launched. 6

10. Although the feature can be disabled, by default some types of error detected by *Mathematica*’s front end will produce an audible beep. For further information about that beep when it occurs, select ‘Why the beep?’ from the HELP menu.

### 8.2 On-Line Help

*Mathematica* provides extensive on-line help. Users can enter a hypertext browser by selecting ‘Wolfram Documentation’ (or some equivalent phrasing) from the HELP menu. The browser can be used to search for an item by entering the name of the item in the text entry box at the top of the browser and then clicking ML on the magnifying glass next to the text entry box; presently, a description of the item will appear. Alternatively, we can navigate the hierarchy of help messages by opening various submenus represented by colored boxes in the window that pops up when ‘Wolfram Documentation’ is selected.

Alternatively, information about a particular item can be obtained simply by typing a question mark ?—which must be the first character on the line—followed by the name of the item about which information is sought (e.g., ?Sin) whenever *Mathematica* is prepared to accept input. This action brings up an inline brief description of the specified item, including a link—the symbol >>—whose selection will bring up the full documentation for the item sought. Beginning a line with two question marks, e.g. ??Sin, appends to the message produced by one question mark a statement about the attributes assigned to the specified symbol.

Finally, when a specific *Mathematica* symbol is highlighted in a notebook, selecting ‘Find Selected Function’ from the HELP menu will bring up the full documentation for that function.

Items in the on-line help library that describe specific *Mathematica* 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 and links to further details and related topics at other points in the on-line messages. Many of the examples contain active statements that can be edited and reexecuted from within the help message. Your attention is drawn particularly to the on-line help messages for the commands D, Integrate, Simplify, Expand, Factor, Solve, Sum, and DSolve.

---

6In the NBI on any platform of course, this feature is unnecessary, since we can easily use resources of the desktop to execute other programs without exiting from *Mathematica*. Indeed, any output produced by Run[ "OSCommand" ] executed from the NBI will be displayed in a pop-up Shell window which, however, will promptly disappear when the command ends.
8.3 Basic Entities in Mathematica

At the very beginning, we point out that Mathematica distinguishes among a variety of different entities, including

- expressions, lists, and character strings, which are the primary quantities with which Mathematica works;
- commands and 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;
- system variables, e.g., $\text{HomeDirectory}$, $\text{IgnoreEOF}$, $\text{Packages}$, $\text{Path}$, $\text{RecursionLimit}$, $\text{TextStyle}$, and $\text{TopDirectory}$, whose values—default or user-specified—control the behavior of various system functions or record various pieces of information about the session. Some of these variables are maintained by Mathematica and are beyond the reach of individual users; others can be customized to the user’s specific ends.
- constants, including $\text{Pi}$ for $\pi$, $\text{I}$ for $\sqrt{-1}$ (which, except in exponents, will be printed in Mathematica output with a somewhat thicker vertical line but which we will display in output more simply as the lower case Roman character $i$), $\text{Infinity}$ for $+\infty$, $\text{E}$ for the base of the natural logarithm (which, except in exponents, will be printed in Mathematica output with an extra vertical line but which we will display in output more simply with the lower case Roman character $e$), and $\text{True}$ and $\text{False}$ for the two logical values. special symbols, including $+$ (addition), $-$ (subtraction), $*$ (multiplication), $/$ (division), $\text{:=}$ (function definition), $\text{/}.$ (perform replacement), $\text{==}$ (equality), and $\text{\rightarrow}$ (transformation rule).
- options, e.g., $\text{Assumptions}$, $\text{PlotRange}$, $\text{TextStyle}$, and $\text{TargetFunctions}$, which may appear among the arguments to functions, may be assigned values, and—when present—modify the output produced by the functions. A full list of the options available, including the default value of each, for a particular function will be displayed when the statement $\text{Options[Function]}$, e.g., $\text{Options[Plot]}$ is executed.

Consistent with Mathematica structure, many of these entities are represented internally in the standard form consisting of a head followed by arguments in the form of a list. The command $\text{FullForm}$, which will be introduced in Section 8.8.2, provides a convenient means to display the internal form of these—and many other—Mathematica objects.

8.4 Variable Names in Mathematica

Legal variable names begin with an alphabetic character and cannot coincide with reserved words like $\text{True}$, $\text{Sin}$, and $\text{Factor}$ but are otherwise unrestricted. They can contain alphabetic and numeric characters but not the underscore character. Within reason, they can be of any length. Lower-case and upper-case letters are treated as distinct. Typing $\langle$\text{ESCAPE}$\rangle$ (or $\backslash$) followed by the spelled-out name of a Greek letter followed by $\langle$\text{ESCAPE}$\rangle$ (or $\backslash$) in an input line will result in the display of the lower-case or upper-case Greek letter both in the input line and in the associated output line, e.g., $\langle$\text{ESCAPE}$\gamma$\text{ESCAPE}$\rangle$ (or $\backslash\text{Gamma}$) in an input line will produce $\gamma$ in the input and output lines while $\langle$\text{ESCAPE}$\Gamma$\text{ESCAPE}$\rangle$ (or $\backslash\text{CapitalGamma}$) will produce $\Gamma$.

Note particularly that, in almost all situations, Mathematica will assume by default that any variable to which no explicit value has been assigned represents a complex quantity.

$^7$Since the names for built-in Mathematica objects (commands, special variables, options, ...) begin with an upper-case letter, avoiding upper-case letters in user-defined variable names will avoid naming conflicts.

$^8$This character is reserved in Mathematica for another purpose, to be outlined in Section 8.6.

$^9$The actual maximum length depends on the operating system, but is quite large even in the most restrictive case.
Expressions are written in Mathematica using a syntax very much like that used in ordinary algebra and are supplied merely by typing the expression at Mathematica's prompt for input,\textsuperscript{10,11} e.g.,

\begin{verbatim}
In[12]:= x + a * Cos[b*x]
\end{verbatim}

to which, when the expression is entered by typing (SHIFT/ENTER),\textsuperscript{12} Mathematica will respond\textsuperscript{13}

\begin{verbatim}
Out[12]= x + a Cos[b x]
\end{verbatim}

The symbols $+$, $-$, $\ast$, $/$, and $^\wedge$ are used for addition, subtraction, multiplication, division, and exponentiation.\textsuperscript{14} Except for those in quoted strings,\textsuperscript{15} spaces are ignored and may be freely inserted for legibility. Functions available for use in expressions include

- General functions, e.g., \texttt{Abs[]}, \texttt{N[]}, \texttt{Trunc[]}, \texttt{Round[]}, \texttt{Sqrt[]}, \texttt{Sign[]}, \texttt{Random[]}, and \texttt{\[]}.

- Trigonometric, hyperbolic and exponential functions and their inverses, e.g., \texttt{Sin[]}, \texttt{Cos[]}, \texttt{Tan[]}, \texttt{ArcSin[]}, \texttt{ArcCos[]}, \texttt{ArcTan[]}, \texttt{Sin[]}, \texttt{Cos[]}, \texttt{Tanh[]}, \texttt{ArcSin[]}, \texttt{ArcCos[]}, \texttt{ArcTan[]}, \texttt{Exp[]}, and \texttt{Log[]}. Note that each function name is capitalized and the argument(s) of the function is (are) enclosed in square brackets.

- 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 Mathematica manuals. A few will be introduced in the remainder of this chapter.

Internally, Mathematica regards a simple expression to be composed of a \textit{head}, which conveys the operator involved, followed by a list of the \textit{operands} known individually as \texttt{part one}, \texttt{part two}, \ldots. (The head can also be referred to as \texttt{part zero}.) Thus, for example, the two expressions

\begin{verbatim}
a + b + c + d  and  abc
\end{verbatim}

are stored internally as the expression

\begin{verbatim}
Plus[ a, b, c, d ]  and  Times[ a, b, c ]
\end{verbatim}

\textsuperscript{10}In this section, we use non-sequential statement numbers to emphasize that many statements may intervene between those illustrated. These statements will all be executable as written, but your statement numbers will almost certainly differ from those presented here.

\textsuperscript{11}In the TBI, the characters \texttt{In[12]:=} will be displayed as a prompt before the statement is typed. In the NBI, no prompt is displayed; the characters—best called a label in this case—will be added after the statement has been submitted for execution.

\textsuperscript{12}See item 4 in Section 8.1 for alternatives. We shall make no further mention of the need for an appropriate termination to each statement.

\textsuperscript{13}Note that we will occasionally rearrange the display of Mathematica output lines to improve legibility. No substance is lost in that rearrangement.

\textsuperscript{14}Mathematica actually allows omission of the symbol \texttt{\*} for conveying multiplication. Thus, the statement \texttt{x \* a \* Cos[b \* x]} is fully equivalent to the one illustrated in line \texttt{In[12]}. Spaces separating variables to be multiplied are then critical, since \texttt{xy} would be interpreted as the two-character name of a (single) variable, not as the product of two variables \texttt{x} and \texttt{y}. While recognizing the simpler appearance of statements in which the asterisk is omitted, we will throughout this book nonetheless note the multiplication explicitly.

\textsuperscript{15}and those separating entities to be multiplied; see previous footnote.
respectively. More complicated expressions are stored as nested simple expressions. Thus, for example, the expressions

\[ a + b - c + d \quad \text{and} \quad a + b(c + d) + e \]

have the internal representations

\[
\text{Plus}\[a, b, \text{Times}[-1, c], d\] \quad \text{and} \quad \text{Plus}[a, \text{Times}[b, \text{Plus}[c, d]], e]
\]

respectively.

As a final example, which anticipates a later need, suppose the expression of interest is the product of three binomial factors, e.g.,

\[(x + 3)^2 (x - a)(x + a)\]

For this expression, we would generate \textit{Mathematica}'s internal representation in several steps finding

\[
\text{Times}[(x+3)^2, x-a, x+a]
\]

\[
\text{Times}[\text{Power}[x+3, 2], \text{Plus}[-a, x], \text{Plus}[a, x]]
\]

\[
\text{Times}[\text{Power}[\text{Plus}[3, x], 2], \text{Plus}[\text{Times}[-1, a], x], \text{Plus}[a, x]]
\]

Here, the first line shows the head and parts 1, 2, and 3 of the original expression while the second line reveals that part 1 of the original expression is itself a structure having a head (\text{Power}; the head of part 1 of the original expression), part 1 (\text{x+3}; part 1 of part 1), and part 2 (2; part 2 of part 1). 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 \text{x} while part 1 of part 2 of part 2 of the original expression is -1. 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.\footnote{The third line conveys the representation actually used in \textit{Mathematica}. Unfortunately, \textit{Mathematica} does not always store the various parts in the same order in which they were entered. After all, the expressions \text{Plus}[a, b, c] and \text{Plus}[b, a, c] are absolutely equivalent in value. However an expression is entered, \textit{Mathematica} chooses a standard order to facilitate subsequent manipulations. Presently, we shall encounter the \textit{Mathematica} command \text{FullForm}, which permits us to instruct \textit{Mathematica} to display its internal representation.}

### 8.6 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. \textit{Mathematica} provides for this shorthand in two ways. First, as a session proceeds, \textit{Mathematica} maintains an internal record of the contents of each \texttt{Out} line, identifying each with the number \texttt{n} that appears in square brackets after the word \texttt{Out}. The item at line \texttt{Out[n]} can then be referred to in subsequent statements either by the label \texttt{Out[n]} or by the label \texttt{\%n}. For example, a later request that \textit{Mathematica} square the expression on line \texttt{Out[12]} in the previous section could be communicated to \textit{Mathematica} either with the statement

\[
\text{In}[18]:= \%12^2
\]

\[
\text{Out}[18]= (x + a \cos [b x])^2
\]

or with the statement
Alternatively, if 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,

\begin{verbatim}
In[20]:= bigexp = x + a * Cos[b*x]
Out[20]= x + a Cos[bx]
\end{verbatim}

for example, associates the variable \texttt{bigexp} with the expression following the assignment 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 \texttt{bigexp}, we might later have written

\begin{verbatim}
In[30]:= bigexp^2
Out[30]= (x + a Cos[bx])^2
\end{verbatim}

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 in a statement like

\begin{verbatim}
In[47]:= bigexp
Out[47]= x + a Cos[bx]
\end{verbatim}

or, alternatively, by interrogating the system with a statement like

\begin{verbatim}
In[48]:= ?bigexp
Global 'bigexp'
bigexp = x + a Cos[bx]
\end{verbatim}

Further, if we can’t remember whether we have already assigned a value to a particular variable, we can invoke the statement

\begin{verbatim}
In[75]:= Names[ "bigexp" ]
Out[75]= { bigexp }
\end{verbatim}

to find out.\footnote{The statement will return a null list (\texttt{\{\}}) if no value has been assigned to the specified variable. Note that the wildcard \texttt{*} can be used in the string. Further, the statement \texttt{Names[]} will return a (long) list of all the symbols known, including those that \texttt{Mathematica} defines for its own purposes.}

Finally, when we have no further need for a particular variable and wish to tell \textit{Mathematica} to forget that we had once bound a value to it, we could enter either of the statements

\begin{verbatim}
In[97]:= Clear[ bigexp ]
In[98]:= Remove[ bigexp ]
\end{verbatim}

the first of which removes any value or definition assigned to the named variable but leaves the variable (and its attributes) known to \textit{Mathematica}\footnote{Alternatively, the shorter statement \texttt{bigexp\textasciitilde} will achieve the same end.} while the second of which completely deletes not only any assigned value or definition but also the name itself so the variable is no longer known to \textit{Mathematica} in any way.
When special functions of significance to a particular session would be useful, we might exploit the Mathematica operator := and the underscore suffix to a variable name\(^{19}\) to append one or more user-defined functions to the built-in functions. For example, the statements

\begin{verbatim}
In[123]:= spectrig[x_] := 1 + Sin[x]^2
In[124]:= mode[x_,y_] := Sin[Pi*x] * Sin[2*Pi*y]
\end{verbatim}

define the functions spectrig(x) and mode(x,y) for subsequent use just as we would use any of Mathematica’s built-in functions.\(^{20}\) Further, if we need to be reminded of the function definition given to a variable, we need simply enter statements like

\begin{verbatim}
In[170]:= ?spectrig
In[171]:= ?mode
\end{verbatim}

Finally, when we have no further need for a particular function and wish to tell Mathematica to forget that we had once associated a function with a particular variable, we would enter an appropriate one or two of the statements

\begin{verbatim}
In[201]:= Clear[ spectrig ]
In[202]:= Remove[ spectrig ]
In[203]:= Clear[ mode ]
In[204]:= Remove[ mode ]
\end{verbatim}

to remove the function definitions or the definitions and their names from Mathematica’s “memory”.

Within Mathematica, some potential variable names are protected. The names Pi (for π), EulerGamma (for Euler’s constant, 0.577216), and 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

\begin{verbatim}
In[227]:= Pi = 4.9
... Set: Symbol π is Protected.
Out[227]= 4.9
\end{verbatim}

though the content of the message may vary with the version of Mathematica. Further, in the illustrated example, both the symbol ‘...’ and the word ‘Set’ are hot keys that give access to additional information.

In rare circumstances, we may wish to use the (protected) symbol anyway. If so, we must unprotect it before using it in the new context, as illustrated in the statements

\begin{verbatim}
In[240]:= Unprotect[ Pi ]; Pi = 4.9
\end{verbatim}

Once we are finished with the non-standard use, we can restore the original status of the variable with the statements

\begin{verbatim}
In[256] := Pi = .; Protect[ Pi ];
\end{verbatim}

\(^{19}\)This suffix, which is used only on the left-hand side of a function definition, flags the variable so that Mathematica will understand that whatever expression, however complicated, appears as the argument of the function is to be substituted for the variable without the suffix on the right-hand side in the function definition.

\(^{20}\)Note the use entirely of lower-case letters in naming the user-defined functions. By adhering to this convention, we avoid conflict with Mathematica’s reserved words.
where the first statement removes the non-standard definition, restoring the default value, and the second statement reestablishes the protection of the variable. Tampering with specially defined variables and other reserved words, however, is risky, and should be undertaken only occasionally and only for good reason.

8.7 Creating and Examining Lists

We have already mentioned in Section 8.3 that Mathematica works with several different types of object, and we have in Sections 8.5 and 8.6 described how to create expressions and assign values to variables. To simplify the quick tour of selected capabilities to be presented in Section 8.8, we include here a description of how lists and lists of lists can be created and examined.

A *list* is an ordered, comma-separated sequence of elements that is enclosed in braces {...}. Lists are created with statements like

```
In[301]:= lst1 = {a, b, a, d, e}
Out[301]= {a, b, a, d, e}
```

```
In[302]:= lst2 = {"David", 5.00, a+b}
Out[302]= {"David", 5.00, a+b}
```

An entire list can be referred to by its name (e.g., `lst2`), while a particular element in the list can be referred to by placing its index in *double* square brackets after the name of the list (e.g., `lst2[[2]]`).

Mathematica contains a few special commands to facilitate the creation of commonly needed lists, including the statements

```
In[303]:= Range[5]
Out[303]= {1, 2, 3, 4, 5}
```

```
In[304]:= Range[5, 10]
Out[304]= {5, 6, 7, 8, 9, 10}
```

```
In[305]:= Range[1, 2, 1/4]
Out[305]= {1, 5/4, 7/4, 2}
```

We draw your attention also to the command `Table`, which can be used to create a list containing values that can be calculated from a formula. For example, the statement

```
In[306]:= Table[n^2, {n, 4}]
Out[306]= {1, 4, 9, 16}
```

creates a list containing the squares of the integers from 1 to 4 while the statement

```
In[307]:= Table[Sin[Pi*n/2], {n, 6}]
Out[307]= {1, 0, -1, 0, 1, 0}
```

creates a list of the values of sin(nπ/2) for n ranging in integer steps from 1 to 6.

Lists of lists are also easily created. We simply place several lists, each of which is enclosed in braces, as the elements of a larger list. The statement

```
In[308]:= lst = {{1, 2, 3}, {4, 5, 6}}
Out[308]= {{1, 2, 3}, {4, 5, 6}}
```

Alternatively, a particular element of a list can be extracted with the function `Part` to be introduced presently. The element `lst2[[2]]` can also be identified as `Part[lst2, 2]`.21
In[308]:= lstlst = {{1,2,3}, {4,5,6}, {7,8,9}}

for example, produces the output

Out[308]= {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}

In essence, the statement at In[308] creates a list of three elements, each of which is itself a list of three elements. The elements of this list of lists can be extracted with statements like

In[309]:= lstlst[[ 2 ]]
Out[309]= {4, 5, 6}

In[310]:= lstlst[[2,3]]
Out[310]= 6

Extract second element of lstlst.

Extract third element of second element of lstlst.

Further, since a list of lists can be viewed as a two-dimensional matrix, Mathematica provides the statement

In[311]:= lstlst // MatrixForm
Out[311]//MatrixForm =

\[
\begin{pmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9
\end{pmatrix}
\]

or MatrixForm[ lstlst ].

to display the list of lists as a matrix. We note that the element lstlst[[1]] is the first row of the matrix, the element lstlst[[2]] is the second row of the matrix, ... and, for example, the element lstlst[[2,3]] is the element in the second row and third column. In other words, in the expression lstlst[[i,j]], the first index i is the row index and the second index j is the column index. Note that the indices of elements in a list are enclosed in double square brackets.

As with simple lists, Mathematica also has a few commands to facilitate creation of special matrices. In particular, you should be aware of the statements

In[312]:= IdentityMatrix[ 4 ]
Out[312]= {{1, 0, 0, 0}, {0, 1, 0, 0}, {0, 0, 1, 0}, {0, 0, 0, 1}}

In[313]:= DiagonalMatrix[{e,f,g,h}]
Out[313]= {{e, 0, 0, 0}, {0, f, 0, 0}, {0, 0, g, 0}, {0, 0, 0, h}}

for creating identity and diagonal matrices and the statements

In[314]:= Table[ Sin[m*x]*Sin[n*y], {m,3}, {n,3} ];
In[315]:= MatrixForm[ % ]

Out[315]//MatrixForm =

\[
\begin{pmatrix}
\sin(x) \sin(y) & \sin(x) \sin(2y) & \sin(x) \sin(3y) \\
\sin(2x) \sin(y) & \sin(2x) \sin(2y) & \sin(2x) \sin(3y) \\
\sin(3x) \sin(y) & \sin(3x) \sin(2y) & \sin(3x) \sin(3y)
\end{pmatrix}
\]

In[316]:= Quit[]

for creating and displaying a matrix whose mn element is \( \sin(mx) \sin(ny) \).

8.8 Using Mathematica . . .

At base, a session with Mathematica involves entering one or more expressions and then instructing Mathematica to process these expressions in some way. The following transcripts of Mathematica
8.8. USING MATHEMATICA ... sessions illustrate some of the simpler manipulations. The transcripts are presented in two columns, the one on the left recording the statements submitted to Mathematica and Mathematica’s responses, the one on the right containing explanatory comments.

Note that the manner of editing statements submitted to Mathematica depends on whether the statement is entered in the TBI or the NBI. In the TBIs on some—but not all—platforms, the arrow keys can be used to retrieve previous statements and move the cursor within a statement, the backspace and delete keys can be used to remove characters, and other keys can be used to insert characters at the position of the cursor. Further—again in some TBI’s, input lines already existing on the screen can be highlighted and then copied to the current input point.

In the NBI, 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 arrow key 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, 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, whichever 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 (SHIFT/ENTER); the full line will be submitted regardless of the position of the cursor within the line.

To keep the several segments in this section separated from one another, we shall periodically submit the statement Quit[]. In the TBI, this statement will terminate the Mathematica session, so we will then have to restart Mathematica; in the NBI, this statement leaves the session running in the same Notebook window. In both cases, however, the action will clear Mathematica’s workspace and, in particular, reinitialize the statement counter. 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 with a fresh invocation of Mathematica.

Finally, note that, throughout this section, various components of Mathematica (functions, commands, operators, ...) are introduced simply by presenting statements in which they are used, frequently with brief comment. Note also that, in the NBI, various entered components will be grouped together into cells, a grouping conveyed by square brackets along the right edge of the Notebook window; the function of these groupings will be addressed briefly in Section 8.16. Details on all mentioned components and on many others can be found in the Mathematica manuals.

8.8.1 ... for Arithmetic

We illustrate first some of the ways in which Mathematica can be instructed to perform simple arithmetic operations. Consider, for example, the statements

\[
\text{In}[1]:= 2 + 3 \\
\text{Out}[1]= 5
\]

Enter a simple numeric expression, terminating with (SHIFT/ENTER). Mathematica returns a value, doing the arithmetic.

---

\[22\] Input statements will look much the same regardless of the platform on which we are operating. Output is here presented approximately as it would appear in the NBI. In the TBI, the content of the output will be the same but, because that interface is limited to character output, its appearance will be less “pretty”.

---
CHAPTER 8. INTRODUCTION TO MATHEMATICA

In[2]:= 4/6
Out[2]= 2/3

Enter another expression. *Mathematica* does not automatically generate a floating point evaluation but *does* reduce the fraction.

In[3]:= Sqrt[9]
Out[3]= 3

Compute \(\sqrt{9}\). Note the capitalization of *Sqrt* and the *square* brackets enclosing the argument.

In[4]:= Sqrt[12]
Out[4]= \(2\sqrt{3}\)

Compute \(\sqrt{12}\). *Mathematica* simplifies the expression but keeps it exact.

In[5]:= Factorial[9]
Out[5]= 362880

Find \(9! = 9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1\), with *Factorial*.

In[6]:= 9!
Out[6]= 362880

Find 9 factorial with postfix operator !.

In[7]:= N[4/6]
Out[7]= 0.66666666666666666667

Ask for floating value with default number of digits.

In[8]:= 4/6 // N
Out[8]= 0.666667

Ask for floating value with default number of digits with postfix operator //.

In[9]:= N[4/6, 20]
Out[9]= 0.66666666666666666667

Ask for floating value with 20 digits.\(^{23}\)

In[10]:= N[Pi, 32]
Out[10]= 3.1415926535897932384626433832795

Ask for floating value of \(\pi\) with 32 digits.

In[11]:= Quit[ ]
(or Exit[ ])

We have, of course, illustrated only some (+, /, Sqrt, Factorial, !, N, //N) of the operators and functions that might be used to persuade *Mathematica* to manipulate with numbers. You should make it a point to read about these and other commands, functions, options, constants, and operators in the *Mathematica* manuals.

8.8.2 . . . for Algebra

*Mathematica* is, of course, capable of much more than simple arithmetic. We next illustrate how algebraic manipulations can be requested, beginning with the statements

In[1]:= (x+3)^2*(x^2-a^2)
Out[1]= (3 + x)^2(-a^2 + x^2)

Enter an expression. Note that the order of the various pieces in the output may differ from that in the input. Compare the response produced on your system by this statement to the response produced by the statement \((x^2-a^2)*(x+3)^2\).

\(^{23}\)While the statement \(\text{N}[4/6,20]\) yields an evaluation with 20 digits, the statement \(\text{N}[4/6,10]\) will on some platforms yield the same value as is displayed by the statement \(\text{N}[4/6]\). Indeed, on those platforms, \(\text{N}[4/6,n]\) yields the same value as \(\text{N}[4/6]\) for any \(n\) less than 17. Only for \(n \geq 17\) does *Mathematica* display a value with the specified number of digits. This behavior stems from the nature of the communication between the kernel (which does the computing) and the front end (which displays the results).
8.8. USING MATHEMATICA ...

Expand the expression. Again, the order of terms may be platform and interface dependent. Here, the symbol \% refers to the most immediate past output. (\%\% and \%\%\% refer to the penultimate and antepenultimate outputs, respectively. The symbol \%n refers to the output on line Out[\n].)

\[
\text{In}[2]:= \text{Expand}[\%]\\
\text{Out}[2]= -9a^2 - 6a^2x + 9x^2 - a^2x^2 + 6x^3 + x^4
\]

Factor the expression in the previous output, assigning the result to the variable fct.

\[
\text{In}[3]:= \text{fct} = \text{Factor}[\%]\\
\text{Out}[3]= -(a - x)(3 + x)^2(a + x)
\]

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 first and third factors \textit{without expanding the second}. To do so, we will have to carve the expression apart, manipulate with its parts, and then reconstruct the expression. As discussed in Section 8.5, \textit{Mathematica} sees this expression as a structure with nested heads and lists of operands. Indeed, we can persuade \textit{Mathematica} to reveal the first level in that structure by exploiting the \textit{Mathematica} functions \texttt{Head} and \texttt{Part} in the statements\textsuperscript{24}

\[
\text{In}[4]:= \{\text{Head}[\text{fct}], \text{Part}[\text{fct},1], \text{Part}[\text{fct},2], \text{Part}[\text{fct},3], \text{Part}[\text{fct},4]\}\\
\text{Out}[4]= \{\text{Times}, -1, a - x, (3 + x)^2, a + x\}
\]

Alternatively, we could exploit not only the function \texttt{Part} but also \textit{Mathematica}’s ability to construct a loop in the statement

\[
\text{In}[5]:= \text{For}[i=0, i<=4, i++, \text{Print}[\text{Part}[\text{fct},i]]]
\]

\[
\times -1\\
a - x\\n(3 + x)^2\\na + x
\]

to show all operands of the expression at once. (Note that we have also here illustrated that the head of an expression can be referred to as part 0 of the expression.)

Yet another way to reveal how \textit{Mathematica} sees the expression fct internally exploits the command \texttt{FullForm}, as in the statement

\[
\text{In}[6]:= \text{FullForm}[\text{fct}]\\
\text{Out}[6]= \text{Times}[\text{\,\,-1, Plus}\[\text{\,\,a, Times}\[-1,\text{x}\}], \text{Power}\[\text{\,\,Plus}\[\text{\,\,3,\,x\}], 2\], \text{Plus}\[\text{\,\,a,\,x}\]]
\]

Once again, we see that fct is a product of four factors \([-1, a - x, (3 + x)^2, a + x]\), though here we have a fuller breakdown of the factors into their own parts.

Now, using the capabilities of the command \texttt{Part}, we might achieve the entire objective of recasting the expression fct with the single nested statement\textsuperscript{25}

\[
\text{In}[7]:= \text{fct} = \text{Part}[\text{fct},1] * \text{Part}[\text{fct},3] * \text{Expand}[\text{Part}[\text{fct},2] * \text{Part}[\text{fct},4]]\\
\text{Out}[7]= -(3 + x)^2(a^2 - x^2)
\]

\textsuperscript{24}While the number of operands is invariant, their order may be platform, interface, and perhaps context dependent. Compare the parts identified here by the integers 0, 1, 2, and 3 with the parts identified by those integers in your environment.

\textsuperscript{25}We repeat the previous footnote: The order of the factors in fct may be platform, interface, and perhaps context dependent. Thus, the numbers identifying various operands in your environment may differ from those used here. \textit{Beware}!
in which we extract parts 1, 2, 3, and 4 from \texttt{fct}, expand the product of parts 2 and 4, and then reconstruct the expression by multiplying that result by parts 1 and 3. This time, we \textit{have} succeeded in reconstructing the original expression. For comparison, note that this expression has the internal representation

\begin{verbatim}
In[8]:= FullForm[ fct ]
Out[8]= Times[ -1, Power[Plus[3,x],2], Plus[ Power[a,2], Times[-1,Power[x,2]] ] ]
\end{verbatim}

We continue with some additional manipulations. First, using the replacement operator \texttt{/} and the operator \texttt{-} (which ultimately prints as \texttt{→} in the NBI) to specify a \textit{transformation rule}, we set the constant \texttt{a} to the specific value 2 with the statement

\begin{verbatim}
In[9]:= fct /. a -> 2
Out[9]= -(3 + x)^2(4 - x^2)
\end{verbatim}

Alternatively, we might have embedded the transformation rule in the command \texttt{Replace} by using the statement\footnote{Some versions of \textit{Mathematica} prior to Version 4.1 did not admit the third argument in the command \texttt{Replace}.}

\begin{verbatim}
In[10]:= Replace[ fct, a->2, 3 ]
Out[10]= -(3 + x)^2(4 - x^2)
\end{verbatim}

Here the third argument specifies how deeply into the structure of \texttt{fct} the replacement is to be carried. Since the \texttt{a} we wish to replace appears at level 3 in \texttt{fct} (see the output of \texttt{FullForm[fct]}), we must extend the replacement to level 3 to achieve the desired substitution. However we make the replacement, we could then find a partial fraction expansion of the reciprocal of this polynomial with the statement

\begin{verbatim}
In[11]:= eq1 = Apart[ 1/%, x ]
Out[11]= 1/100(-2 + x) - 1/4(2 + x) + 1/5(3 + x)^2 + 6/25(3 + x)
\end{verbatim}

Numerous rewrites of this expression are now possible. For example, replacing \texttt{x} with \texttt{3*z} would be accomplished by using the \texttt{Replace} command in the statement

\begin{verbatim}
In[12]:= Replace[ eq1, x -> 3*z, 4 ]
Out[12]= 1/100(-2 + 3z) - 1/4(2 + 3z) + 1/5(3 + 3z)^2 + 6/25(3 + 3z)
\end{verbatim}

Alternatively, this same replacement could be accomplished with the statement

\begin{verbatim}
In[13]:= eq1 /. x -> 3*z
\end{verbatim}

though we suppress the output, since it is the same as that at line Out[12].

To illustrate a more complicated recasting, suppose that we wanted to square the denominator of the third term in \texttt{eq1} without affecting any other term. The statement \texttt{Expand[eq1]} does nothing. (Try it.) We need a more sophisticated operation that rearranges only the third term (part 3 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. The task takes several steps, beginning with the identification of the proper part with the statements
8.8. USING MATHEMATICA ...

In[14]:= Part[ eq1, 3 ]
Out[14]= 1
5(x + 3)2

In[15]:= Part[ %, 2 ]
Out[15]= 1
(x + 3)2

In[16]:= Part[ eq1, 3, 2 ]
Out[16]= 1
(x + 3)2

Then we expand the denominator of this piece of eq1 and substitute it into eq1 as the second part of the third part with the statements

In[17]:= 1/Expand[1/%]
Out[17]= 1
9 + 6x + x2

In[18]:= ReplacePart[ eq1, %, {3, 2} ]

(Note that the particular part to be replaced (part {3,2}) is conveyed to ReplacePart with a list.

Out[18]= 1
100(−2 + x) − 1
4(2 + x) + 6
25(3 + x) + 1
5(9 + 6x + x2)

In[19]:= Quit[]

We have, of course, only mentioned a few (Expand, Factor, Part, Replace, ReplacePart) of the numerous functions Mathematica makes available for manipulating expressions algebraically. We have also introduced the control structure For, the postfix replacement operator /., and the special symbol -> (which ultimately is displayed as → in the NBI). The differences among several commands, including Expand, ExpandAll, Factor, Simplify, FullSimplify, Together, Collect, Apart, Cancel, Numerator, Denominator, and Factor are explored in one of the exercises. You should make it a point to read about these functions and any associated options in the Mathematica manuals.

8.8.3 ... for Complex Variables

Mathematica is also capable of manipulating complex variables. Indeed, in most contexts, Mathematica assumes that any variable to which no specific value has been assigned represents a complex value. We illustrate some of Mathematica’s main capabilities in this category with the statements

In[1]:= fct = Sin[x+y*I]
Out[1]= Sin[x + i y]

In[2]:= ComplexExpand[ fct ]

Define complex expression. The constant I stands for \(\sqrt{-1}\) and is displayed as i in output.

Recast fct in form \(a + ib\). In contrast to the default behavior, the function ComplexExpand assumes that variables to which no value has been assigned represent real values.
In[3]:= ComplexExpand[ Re[fct] ]

In[4]:= ComplexExpand[ Im[fct] ]

In[5]:= ComplexExpand[ Conjugate[fct] ]

Extract real part of fct.

Extract imaginary part of fct.

Find complex conjugate of fct.

In[6]:= fct1 = (x + I*y)/(x - I*y)
Out[6]= x + i y

In[7]:= ComplexExpand[ fct1, TargetFunctions -> {Re, Im} ]
Out[7]= x^2 + y^2 + 2 i xy

Recast fct1 in form a + ib. ComplexExpand usually tries to express its output in terms of the real and imaginary parts of the input variables. Sometimes, however, the option TargetFunctions must be used to specify which of Re, Im, Abs, Arg, Conjugate and Sign are to be used. We include that option here simply to illustrate it; the option is not actually necessary in this context.

Find complex absolute value of previous result.

Find complex argument of fct1. In Mathematica, \( \text{ArcTan}[a,b] = \tan^{-1}(b/a) \).

To conclude this section, we record quickly how to convert complex numbers from Cartesian form \( z = x + iy \) to polar form \( z = |z| e^{i \text{Arg}(z)} \) and from polar form to Cartesian form. Starting with a complex number in Cartesian form, we would use statements like

In[13]:= fct3 = x + I*y
Out[13]= x + I y

In[14]:= ComplexExpand[ Abs[fct3], TargetFunctions -> {Re, Im} ]
Out[14]= \( \sqrt{x^2 + y^2} \)

In[15]:= ComplexExpand[ Arg[fct3], TargetFunctions -> {Re, Im} ]
Out[15]= \( \text{ArcTan}[x, y] \)

Simplify it, though Mathematica doesn't seem to recognize that ArcTan\( [x^2 - y^2, 2yx] \) would be even simpler.

Find complex absolute value of fct3, noting that this route includes the simplification that we had to request explicitly in the previous route.

And simplify it.
8.8. USING MATHEMATICA ...

\[\text{In}[16]:= fct4 = r \times \exp[\ I\times[\Theta]] \]
\[\text{Out}[16]= e^{i\theta} r \]
\[\text{In}[17]:= \text{ComplexExpand[} \text{ExpToTrig}[fct4] \]
\[\text{Out}[17]= r \cos[\theta] + i r \sin[\theta] \]
\[\text{In}[18]:= \text{Quit[]} \]

to find the Cartesian expression of a complex number originally given in polar form.

We have, of course, only mentioned some (\text{ComplexExpand}, \text{Re}, \text{Im}, \text{Conjugate}, \text{Abs}, \text{Arg}, \text{ExpToTrig}) of the functions \text{Mathematica} makes available for manipulating complex numbers. Further, we have also introduced the option \text{TargetFunctions} for the command \text{ComplexExpand}, and we have seen additional contexts in which the command \text{Simplify} can be valuable. You should make it a point to read about these functions and any associated options in the \text{Mathematica} manuals.

8.8.4 ... for Trigonometry

\text{Mathematica} 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

\[\text{In}[1]:= \text{trig1} = \sin[x] \times \cos[x] \]
\[\text{Out}[1]= \cos[x] \sin[x] \]
\[\text{In}[2]:= \text{TrigToExp[} \text{trig1} \]
\[\text{Out}[2]= \frac{1}{4} e^{-2ix} - \frac{1}{4} e^{2ix} \]
\[\text{In}[3]:= \text{ExpToTrig[} \% \]
\[\text{Out}[3]= \frac{1}{2} \sin[2x] \]
\[\text{In}[4]:= \text{TrigReduce[} \text{trig1} \]
\[\text{Out}[4]= \frac{1}{2} \sin[2x] \]
\[\text{In}[5]:= \text{TrigExpand[} \% \]
\[\text{Out}[5]= \cos[x] \sin[x] \]
\[\text{In}[6]:= \text{trig2} = \sin[2x+y] \]
\[\text{Out}[6]= \sin[2x+y] \]
\[\text{In}[7]:= \text{TrigExpand[} \text{trig2} \]
\[\text{Out}[7]= 2 \cos[x] \cos[y] \sin[x] + \cos[x]^2 \sin[y] - \sin[x]^2 \sin[y] \]
\[\text{In}[8]:= \text{TrigReduce[} \% \]
\[\text{Out}[8]= \sin[2x+y] \]
\[\text{In}[9]:= \text{TrigExpand[} \text{trig2/.x->z/2/.z->2*x} \]
\[\text{Out}[9]= \cos[y] \sin[2x] + \cos[2x] \sin[y] \]
\[\text{In}[10]:= \text{TrigExpand[} \% \]
\[\text{Out}[10]= 2 \cos[x] \cos[y] \sin[x] + 2 \cos[x]^2 \sin[y] - \sin[x]^2 \sin[y] \]

Define a trig expression.
Convert to exponential form.
Convert to standard trig functions.
Recast with multiple angle identities.
Expand functions of multiple angles.
Define another trig function.
Expand fully using trig addition formulae.
Return to original form.
Expand, but don't expand 2x. Here, we replace 2x with z for the expansion but then replace z with 2x after the expansion has been evaluated.
Expand again to complete expansion.
In[7]:=(Cos[x]+Sin[x])^2
Define yet another trig function.
Out[7]= (Cos[x] + Sin[x])^2

In[8]:= Expand[%]
Expand binomial.

In[9]:= Quit[]

We have, of course, only mentioned a few (TrigToExp, ExpToTrig, TrigReduce, TrigExpand, and Expand) of the functions Mathematica makes available for doing trigonometry. You should make it a point to read about these and other commands and any applicable special arguments or options in the Mathematica manuals.

8.8.5 ... for Algebraic Equations

We turn next to illustrating a few Mathematica commands for manipulating expressions algebraically. To define a polynomial and find its roots, for example, we would use statements like

In[1]:= poly = Expand[(x^2-a^2)*(x+3)^2];
Define a polynomial.
In[2]:= soln = Solve[{poly == 0}, {x}]
Solve the equation poly = 0 for x. There are four roots, though two of them are identical.
Out[2]= \\
{\\{x\to-3\\},\\{x\to-3\\},\\{x\to-a\\},\\{x\to a\\}\\}

Note that the arguments to the command Solve are lists, and that the roots have been returned as a list of lists of transformation rules—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

In[3]:= soln1 = Solve[poly == 0, x]
Out[3]= \\
{\\{x\to-3\\},\\{x\to-3\\},\\{x\to-a\\},\\{x\to a\\}\\}

Note also the use of the double equal sign in setting up a logical equality between two quantities as opposed to the assignment of a value to a variable (for which Mathematica uses a single equal sign).

Of course, once we have obtained roots, we should verify their correctness by backsubstitution into the original equation. To do so, we would execute statements like

In[4]:= poly /. x -> -a
Verify first root.
Out[4]= 0
In[5]:= poly /. soln
Verify all roots in a single statement.
Out[5]= {0, 0, 0, 0}

In[6]:= For[i=1, i<=4, i++,
Print[poly /. soln[[i]]]
Verify all roots, using the For structure to code a loop.
0
0
0
0

Perhaps the most famous of all solutions is that of a quadratic equation. The command Solve clearly knows the quadratic formula, as illustrated by the statements
8.8. USING MATHEMATICA ...

Supply a quadratic expression.

In[7]:= a*x^2 + b*x + c;

Solve \( ax^2 + bx + c = 0 \) for \( x \). Note again that the roots have been returned as a list of lists.

In[8]:= Solve[ %==0, x ]

Out[8]=

\[
\left\{ \left\{ x \rightarrow -\frac{-b - \sqrt{b^2 - 4ac}}{2a} \right\}, \left\{ x \rightarrow -\frac{-b + \sqrt{b^2 - 4ac}}{2a} \right\} \right\}
\]

Evaluate solution with indicated values of \( a, b, \) and \( c \).

In[9]:= % /. {a->3, b->4, c->2}

Out[9]=

\[
\left\{ \left\{ x \rightarrow -\frac{1}{6} \left( -4 - 2i \sqrt{2} \right) \right\}, \left\{ x \rightarrow \frac{1}{6} \left( -4 + 2i \sqrt{2} \right) \right\} \right\}
\]

We can, of course, always ask for floating point evaluations with the command \( \text{N} \), finding here that

In[10]:= N[ % ]

Out[10]=

\[
\left\{ \left\{ x \rightarrow -0.666667 + 0.471405 i \right\}, \left\{ x \rightarrow -0.666667 - 0.471405 i \right\} \right\}
\]

We have accepted the default value (6) for the number of digits in the floating point values returned by \( \text{N} \).

Finally, let us illustrate \textit{Mathematica’s} capability to solve simultaneous linear equations. To define and solve a representative system, we invoke the statements

In[11]:= eqn1 = x - y == 4

Give \textit{Mathematica} a pair of linear equations, binding each to a variable to facilitate subsequent reference.


In[12]:= eqn2 = x + y == 0

Solve the system for \( x \) and \( y \).

Out[12]= x + y == 0

In[13]:= Solve[ {eqn1,eqn2}, {x,y} ]

Note that the equations and the variables are presented to \texttt{Solve} as lists and that each solution—here there is only one—is presented as a list. Were there several solutions, \textit{Mathematica} would have returned a list of lists.

\[
\left\{ \left\{ x \rightarrow 2, y \rightarrow -2 \right\} \right\}
\]

We need not, of course, have specific numerical values for the coefficients in the equations to be solved. Solution of a more symbolic system of linear equations is illustrated in the statements

In[14]:= eqn1 = a*x + b*y == e;

Try it with a more symbolic system.

In[15]:= eqn2 = c*x + d*y == f;

In[16]:= Solve[ {eqn1,eqn2}, {x,y} ]

Remove variables eqn1 and eqn2 from \textit{Mathematica’s} memory.

Out[16]=

\[
\left\{ \left\{ x \rightarrow \frac{de - bf}{bc - ad}, y \rightarrow -\frac{-ce + af}{bc - ad} \right\} \right\}
\]

In[17]:= Remove[ eqn1, eqn2 ]

In[18]:= Quit[]

We have, of course, only mentioned one (\texttt{Solve}) of the functions \textit{Mathematica} makes available for solving algebraic equations exactly. Your attention is drawn also to the function \texttt{Nsolve} for finding solutions numerically and to the function \texttt{LinearSolve}, which is built to solve matrix equations of the form \( Ax = b \). You should make it a point to read about these functions in the \textit{Mathematica} manuals. Rootfinding is discussed in greater depth in Chapter 14.
8.8.6 ... for Manipulating Vectors and Matrices

Vectors and matrices represent numerous physical quantities. We have already seen in Section 8.7 how to create lists. Vectors are simply lists and (two-dimensional) matrices are lists of lists. Once they have been created, Mathematica can manipulate these objects in numerous ways. To illustrate some of these capabilities, we begin by defining a matrix and evaluating its transpose with the statements

In[1]:= mat1 = { {1,2}, {3,4} };  
Create mat1.

In[2]:= MatrixForm[ mat1 ]  
Display mat1 in matrix form.

Out[2]//MatrixForm= 
\[
\begin{pmatrix}
1 & 2 \\
3 & 4 \\
\end{pmatrix}
\]

In[3]:= mat2 = Transpose[ mat1 ];  
Evaluate transpose of mat1.

In[4]:= MatrixForm[ mat2 ]  
Display transpose in matrix form.

Out[4]//MatrixForm= 
\[
\begin{pmatrix}
1 & 3 \\
2 & 4 \\
\end{pmatrix}
\]

Then, we evaluate the inverse, the trace, and the determinant of the matrix with the statements

In[5]:= MatrixForm[ Inverse[ mat1 ] ]  
Evaluate inverse of mat1 and display it in matrix form.

Out[5]//MatrixForm= 
\[
\begin{pmatrix}
-2 & 1 \\
3 & -1/2 \\
\end{pmatrix}
\]

In[6]:= val1 = Tr[ mat1 ]  
Evaluate trace (sum of diagonal elements) of mat1.


In[7]:= val2 = Det[ mat1 ]  
Evaluate determinant of mat1.


Other common manipulations include evaluating the standard matrix product and an element-by-element product. The standard (non-commutative) matrix product is specified by the infix operator ‘.’ in a statement like

In[8]:= MatrixForm[ mat1.mat2 ]  
Evaluate the result of ordinary matrix multiplication and display in matrix form.

Out[8]//MatrixForm= 
\[
\begin{pmatrix}
5 & 11 \\
11 & 25 \\
\end{pmatrix}
\]

Element-by-element (commutative) multiplication is accomplished with the infix operator ‘*’ in a statement like

In[9]:= MatrixForm[ mat1*mat2 ]  
Evaluate the result of element-by-element multiplication.

Out[9]= 
\[
\begin{pmatrix}
1 & 6 \\
6 & 16 \\
\end{pmatrix}
\]

Finally (for this section anyway), Mathematica includes functions for finding eigenvalues and eigenvectors, which we illustrate with the statements
In[10]:= spinx = {{0, 1/Sqrt[2], 0}, {1/Sqrt[2], 0, 1/Sqrt[2]}, {0, 1/Sqrt[2], 0}};
Enter a matrix bound to the variable spinx.

In[11]:= MatrixForm[ % ]
Display the matrix in matrix form.

Out[11]=
\[
\begin{pmatrix}
0 & 1 \\
\frac{1}{\sqrt{2}} & 0 \\
0 & \frac{1}{\sqrt{2}}
\end{pmatrix}
\]

In[12]:= Eigenvalues[ spinx ]
Find eigenvalues.

Out[12]= [-1, 1, 0]

In[13]:= Eigenvectors[ spinx ]
Find eigenvectors.

Out[13]= {{1, -\sqrt{2}, 1}, {1, \sqrt{2}, 1}}{-1, 0, 1}

In[14]:= Eigensystem[ spinx ]
Find eigenvalues and eigenvectors with a single statement.

Out[14]= {{-1, 0, 1}, {{1, -\sqrt{2}, 1}, {-1, 0, 1}, {1, \sqrt{2}, 1}}}

In[15]:= MatrixForm[%]
Display result in matrix form, in which the first row contains the eigenvalues and the second row contains the associated eigenvectors.

Out[15]=
\[
\begin{pmatrix}
-1 & 1 & 0 \\
1, -\sqrt{2}, 1 & 1, \sqrt{2}, 1 & -1, 0, 1
\end{pmatrix}
\]

In[16]:= Quit[]

The output of Eigenvalues is a list of the eigenvalues of the specified matrix. The output of Eigenvectors is a list of lists, each of the latter of which is the (unnormalized) eigenvector belonging to the eigenvalue in the same position in the list returned by Eigenvalues. Finally, the output of Eigensystem is a two-element list, the first element of which is the output of Eigenvalues and the second element of which is the output of Eigenvectors.

We have, of course, only mentioned a few (MatrixForm, Transpose, Inverse, Tr, Det, Eigenvalues, Eigenvectors, Eigensystem) of the functions available for manipulating matrices. Important additional functions include AppendColumns, AppendRows, BlockMatrix, TakeRows, TakeColumns, TakeMatrix, SubMatrix, ZeroMatrix, and LinearEquationsToMatrix, which are contained in the add-on package\textsuperscript{27} LinearAlgebra'MatrixManipulation\textsuperscript{4} and the function Normalize, which is contained in the add-on package LinearAlgebra'Orthogonalization\textsuperscript{4}. You should make it a point to read about these functions and any associated arguments in the Mathematica manuals.

8.8.7 . . . for Basic Calculus

Differentiation, indefinite and definite integration, and determination of series representations of symbolic expressions are also among Mathematica’s capabilities. To evaluate derivatives, for example, we would use statements like

\textsuperscript{27} Packages are discussed in Section 8.9.
CHAPTER 8. INTRODUCTION TO MATHEMATICA

Enter an expression.

Differentiate it with respect to \( x \).

Differentiate it twice with respect to \( x \) and simplify the result.

To evaluate integrals, we would instead use statements like

Evaluate indefinite integral of the first derivative.

Enter another expression.

Evaluate definite integral on \( x \) from \( x = 0 \) to \( x = +\infty \).

Interestingly, Mathematica in this case returns a result revealing that the integral is not well defined. In the returned conditional expression the first argument is the evaluation of the integral, but only when the condition in the second argument is true. If the specified condition is false (\( \text{Re}(a)=0 \)), then the denominator in the integrand passes through zero somewhere in the interval of integration and the integrand is indeed ill defined. If, however, the specified condition is true, then the integrand is well defined, having the value \( 1/\sqrt{a^2} \), a result not written as \( 1/a \) because \( a \) is not known to be positive or even real.

We can instruct Mathematica to be more specific by exploiting an optional argument to \texttt{Integrate}, as for example in the statements

Tell Mathematica that \( a \) is real.

Tell Mathematica that \( a \) is positive, which implies as well that \( a \) must be real.

Finally (for this section), we point out that Mathematica can generate series expansions of prescribed functions:

Enter yet another expression.

Expand the expression in a Taylor series in \( x \) about \( x = 0 \), retaining terms through 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.
\begin{verbatim}
In[11]:= FullForm[ ser ]
Out[11]= FullForm[ SeriesData[ x, 0, List[ Rational[1,2], Rational[1,4], 0, Rational[-1,48],
0, Rational[1,480], 0, Rational[-17,80640]], 0, 8, 1 ]]
\end{verbatim}

reveals that the head of an expression that conveys a series is **SeriesData**. The remaining parts of the expression are, in order, the variable \( x \) in which the series is expressed, the value \( x_0 \) (here 0) of \( x \) about which the expansion is made, a list of the coefficients of the successive powers of \( x - x_0 \) in the expansion, and two arguments (here 0 and 8) which convey the starting and ending powers as multiples of the final argument (here 1). The powers appearing thus are \( 0/1 \), \( 2/1 \), \( 3/1 \), ... \( 8/1 \).

Unfortunately, for some operations, the expression of a series in the form dictated by **SeriesData** cannot be used. In those cases, we may need to convert the series into an ordinary expression by invoking the command **Normal** with the statement

\begin{verbatim}
In[12]:= pol = Normal[ ser ]
Out[12]= 1/2 + x^4 - x^3/48 + x^5/480 - 17x^7/80640

In[13]:= FullForm[ pol ]
Out[13]= Plus[ Rational[1,2], Times[ Rational[1,4], x], Times[ Rational[-1,48], x^3],
Times[ Rational[1,480], x^5], Times[ Rational[-17,80640], x^7] ]

In[14]:= Quit[]
\end{verbatim}

Note that the omitted terms in the series are not indicated in \( \text{pol} \) and that \( \text{pol} \) has expression type **Plus**. Further, compare the storage of a series as revealed by the statement **FullForm[ser]** with the storage of a polynomial as revealed by the statement **FullForm[pol]**.

By default, **Mathematica** commands automatically carry out particular operations that, in **Mathematica**’s mind, simplify expressions submitted to them. As we have seen above, the command **D** for differentiation, for example, will result in immediate evaluation of any derivatives that can be evaluated. Sometimes, however, we may wish to enter an expression and suppress immediate evaluation. **Mathematica** provides the command **HoldForm** for that purpose and the command **ReleaseHold** to request explicit evaluation at a later time. To illustrate, suppose we define the function

\begin{verbatim}
In[1]:= f[y_] := a * Sin[ \[Omega] * y ]
\end{verbatim}

Then, the two statements

\begin{verbatim}
In[2]:= D[ f[y], y ]

In[3]:= d1 = HoldForm[ D[ f[y], y ] ]
Out[3]= \[PartialD]_y f[y]
\end{verbatim}

reveal the difference. 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}
In[47]:= ReleaseHold[ d1 ]
Out[47]= a\[Omega] Cos[ y\[Omega] ]
\end{verbatim}
This feature might, for example, be useful if we had several functions whose derivative we wished to evaluate. We would simply define the derivative with `HoldForm`. Then, we could evaluate the derivative of each new function by changing the definition of \( f[y] \) before executing the simple statement illustrated on Line `In[47]`.

\(\text{In}[48]:=\quad \text{Quit[]}\)

We have, of course, only mentioned a few (\(D, \text{Integrate}, \text{Series}, \text{Normal}\)) of the numerous functions *Mathematica* makes available for doing calculus. One important additional function is \(\text{Limit}\). You should make it a point to read about these functions and any associated options in the *Mathematica* manuals.

### 8.8.8 ... for Laplace Transforms

In Section 1.5.2, we defined the Laplace transform \(\tilde{f}(s)\) of a function \(f(t)\) by the integral

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

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.

*Mathematica* provides for quick evaluation of an assortment of integral transforms. In particular the function `LaplaceTransform` will calculate a Laplace transform and the function `InverseLaplaceTransform` will invert a transform to return to the original space. We illustrate the use of those functions with the statements

\(\text{In}[1]:=\quad \text{Sin}\left[\Omega t\right] ;\)

\(\text{In}[2]:=\quad \text{LaplaceTransform}\left[ \%, t, s \right] \quad \text{Out}[2]=\quad \frac{\omega}{s^2 + \omega^2}\)

\(\text{In}[3]:=\quad \text{InverseLaplaceTransform}\left[ \%, s, t \right] \quad \text{Out}[3]=\quad \text{Sin} \left[ t \Omega \right]\)

\(\text{In}[4]:=\quad \text{LaplaceTransform}\left[ \text{Exp}\left[ -a t \right] , t, s \right] \quad \text{Out}[4]=\quad \frac{1}{a + s}\)

\(\text{In}[5]:=\quad \text{Simplify}\left[ \text{InverseLaplaceTransform}\left[ \frac{1}{(s+a)(s+b)} , s, t \right] \right] \quad \text{Out}[5]=\quad \frac{e^{-at} - e^{-bt}}{a - b}\)

\(\text{In}[6]:=\quad \text{Quit[]}\)
8.8. USING MATHEMATICA...

8.8.9 . . . for Ordinary Differential Equations

Among Mathematica’s strongest suits is its ability to solve ordinary differential equations (ODEs). We will return in Chapter 11 to a more extended discussion of those capabilities. For this chapter, we limit ourselves to a quick illustration, choosing 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 Mathematica with the statement

\[
\text{In}[1]:= \text{deq1} = m x''[t] + b x'[t] + k x[t] == 0
\]

\[
\text{Out}[1]= k x[t] + b x'[t] + m x''[t] == 0
\]

To solve this equation for \( x \) as a function of \( t \), we can proceed in either of two ways. If we submit the statement

\[
\text{In}[2]:= \text{soln1} = \text{DSolve}[\text{deq1}, x[t], t]
\]

Mathematica returns the solution in the form

\[
\text{Out}[2]= \left\{ x[t] \to e^{(-b-\sqrt{b^2-4km})t/2m} C[1] + e^{(-b+\sqrt{b^2-4km})t/2m} C[2] \right\}
\]

Here, \( \text{DSolve} \) has used the symbols \( C[1] \) and \( C[2] \) for the initially undetermined integration constants in the solution to a second-order differential equation. Alternatively, we might phrase the invocation of \( \text{DSolve} \) in the form

\[
\text{In}[3]:= \text{soln2} = \text{DSolve}[\text{deq1}, x[t], t]
\]

in which case Mathematica returns the solution in the form

\[
\text{Out}[3]= \left\{ x[t] \to \text{Function}[\{t\}, e^{(-b-\sqrt{b^2-4km})t/2m} C[1] + e^{(-b+\sqrt{b^2-4km})t/2m} C[2]] \right\}
\]

where again the symbols \( C[1] \) and \( C[2] \) have been used for the integration constants.

The two forms of the solution look similar. Each is expressed as a transformation rule. The solution \( \text{soln1} \), however, is not a function. The rule expressing \( \text{soln1} \) allows substitution of the solution for \( x[t] \) in any expression. For example, the statement

\[
\text{In}[4]:= x[t] + D[x[t], t] /. \text{soln1}
\]

\[
\text{Out}[4]= \left\{ e^{(-b-\sqrt{b^2-4km})t/2m} C[1] + e^{(-b+\sqrt{b^2-4km})t/2m} C[2] + x'[t] \right\}
\]

yields a result in which \( x[t] \) has been replaced in the first term but the derivative has been left unevaluated. At times, this first form of the solution will be adequate. If, however, we expect to manipulate with the solution, the second form is preferable, since it expresses the solution as a true function of the independent variable \( t \). Had we used \( \text{soln2} \) in the example at \( \text{In}[4] \), i.e., had we submitted the statement

\[
\text{In}[5]:= x[t] + D[x[t], t] /. \text{soln2}
\]

\[
\text{Out}[5]= \left\{ e^{(-b-\sqrt{b^2-4km})t/2m} C[1] + e^{(-b+\sqrt{b^2-4km})t/2m} C[2] + x'[t] \right\}
\]

\[28\] When the variable with respect to which a derivative is to be evaluated is the only quantity on which a function depends, we can use a prime ‘’ to convey differentiation rather than the more explicit form \( D[x[t], t] \).

\[29\] As is often the case, we here present the output in a form that is more transparent than—i.e., different from—the form presented directly in the Mathematica output. In what follows, if we extract explicit parts of the expressions, we will do so in ways that do not depend on these differences.
Mathematica would have returned a more complicated result in which the solution would have been substituted both in the first and in the second term and then the derivative would have been evaluated.

Whichever form of the solution we have requested, Mathematica’s response to the simple statements we have used has been blindly to write a solution that is mathematically correct but has not recognized that the most convenient form of this solution will depend on the algebra 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 Mathematica 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 Mathematica that \( b^2 < 4km \), an end that is (unfortunately) not easily accomplished, partly because Mathematica assumes that variables to which no value has been assigned represent complex quantities and does not have a facility for overriding that assumption except within the context of a few specific commands. The following sequence of statements will achieve the desired objective.\(^{30}\) We begin by using the statement

\[
\text{In}[6]:= \text{soln3 = soln1[[1]][[1]][[2]]}
\]

\[
\text{Out}[6]= e^{(-b - \sqrt{b^2 - 4km})t/2m} C[1] + e^{(-b + \sqrt{b^2 - 4km})t/2m} C[2]
\]

to extract the second part of the first part of the first part of \text{soln1}, i.e., to extract an expression giving the solution itself. Then, we simplify the appearance of this expression by introducing (temporarily) the variables

\[
\gamma = \frac{b}{2m} \quad \text{and} \quad \omega = \sqrt{\frac{k}{m}} \quad (8.2)
\]

with the statement

\[
\text{In}[7]:= \text{soln4 = soln3 /\{b -> 2*\[Gamma]\*m, k -> \[Omega]^2*m \}}
\]

\[
\text{Out}[7]= e^{(-2m\gamma - \sqrt{4m^2\gamma^2 - 4m^2\omega^2})/2m} C[1] + e^{(-2m\gamma + \sqrt{4m^2\gamma^2 - 4m^2\omega^2})/2m} C[2]
\]

Next, we invoke the statement\(^{31}\)

\[
\text{In}[8]:= \text{soln5 = Simplify[ soln4, m > 0 ]}
\]

\[
\text{Out}[8]= e^{-t(\gamma + \sqrt{\gamma^2 - \omega^2})} \left( C[1] + e^{(2t\sqrt{\gamma^2 - \omega^2})} C[2] \right)
\]

to clear the \( m \)'s, the statement

\[
\text{In}[9]:= \text{soln6 = Expand[ soln5 ]}
\]

\[
\text{Out}[9]= e^{-t(\gamma + \sqrt{\gamma^2 - \omega^2})} C[1] + e^{(2t\sqrt{\gamma^2 - \omega^2}) - t(\gamma + \sqrt{\gamma^2 - \omega^2})} C[2]
\]

\(^{30}\) The author continues to search for less cumbersome routes but, so far anyway, has failed to find any.

\(^{31}\) Some versions of Mathematica prior to Version 4.1 did not admit the second argument in the command \text{Simplify}.  

to distribute the initial factor over both terms, and the command \texttt{Map} in the statement

\texttt{In[10]:= soln7 = Map[ Expand, soln6, 3 ]}

\texttt{Out[10]= e^{-t\gamma -t\sqrt{\gamma^2 - \omega^2}}C[1] + e^{-t\gamma +t\sqrt{\gamma^2 - \omega^2}}C[2]}

\texttt{Out[10]= e^{-t\gamma -t\sqrt{\gamma^2 - \omega^2}}C[1] + e^{-t\gamma +t\sqrt{\gamma^2 - \omega^2}}C[2]}

We take the next step with the statement

\texttt{In[12]:= soln9 = Factor[ Map[ ComplexExpand, soln8 ] ]}

\texttt{Out[12]= e^{-t\gamma} \left( C[1] \cos(at) + C[2] \cos(at) - iC[1] \sin(at) + iC[2] \sin(at) \right)}

From this solution, however, we note that the expression would have a yet simpler form if we substituted $A$ for $C[1] + C[2]$ and $B$ for $-i(C[1] - C[2])$. We achieve that objective with the statements


\texttt{Out[13]= \{\{C[1] \to \frac{1}{2}(A+iB), C[2] \to \frac{1}{2}(A-iB)\}\}}

\texttt{In[14]:= soln10 = Simplify[ soln9 /. %[[1]] ]}

\texttt{Out[14]= e^{-t\gamma} \left( A \cos(at) + B \sin(at) \right)}

We, of course, are not finished until we have imposed suitable initial conditions on the solution. To do so, we determine $A$ and $B$ to reflect the simple assumption that the motion is started by drawing the oscillator aside and releasing it from rest, i.e., that

$$x(0) = x_0; \quad \frac{dx}{dt}(0) = 0$$

Then, we find the constraints imposed on the integration constants with the statements

\texttt{In[15]:= eq1 = x0 == soln10 /. t -> 0}

\texttt{Out[15]= x0 == A}

\texttt{In[16]:= eq2 = 0 == D[ soln10, t ] /. t -> 0}

\texttt{Out[16]= 0 = aB - A\gamma}

\texttt{In[17]:= Solve[ \{eq1,eq2\}, \{A,B\} ]}

\texttt{Out[17]= \{\{B \to \frac{x_0\gamma}{a}, A \to x_0\}\}}

\texttt{In[18]:= soln11 = soln10 /. %[[1]]}

\texttt{Out[18]= e^{-t\gamma} \left( x_0 \cos(at) + \frac{x_0\gamma \sin[at]}{a} \right)}

\texttt{In[19]:= soln12 = Collect[ soln11, x0 ]}

\texttt{Out[19]= e^{-t\gamma} x_0 \left( \cos(at) + \frac{\gamma \sin[at]}{a} \right)}
To interpret this result in terms of the original dimensional parameters, we need merely remember that

\[ a = \sqrt{\omega^2 - \gamma^2} \quad ; \quad \gamma = \frac{b}{2m} \quad ; \quad \omega^2 = \frac{k}{m} \]  

(8.4)

These values could, of course, be reinserted into the solution obtained at line Out[19]. At long last, we have reached a form of the solution that reveals its character as an oscillation with an exponential decay of its amplitude. We leave it to the reader to use Mathematica to verify that this solution in fact satisfies the original equation and the imposed initial conditions. (Similar simpler expressions for the critically and overdamped cases are deduced in the exercises.)

In the above example, we chose to leave the initial conditions unspecified until after the solution had been obtained and recast in a more transparent form. Alternatively, we could have incorporated the initial conditions in the original invocation of DSolve with an expanded statement of the form

\[
\text{In}[20]:= \text{soln}13 = \text{DSolve}\{\text{deq1}, x[0]==x0, x'[0]==0 \}, x[t], t \}
\]

The resulting solution would still have been presented in an exponential form in which, for the underdamped case, the square roots appearing would have been imaginary. We would still have to work our way through a number of transformations before having the solution in the standard form for the underdamped case. Note specifically that the equations and initial conditions have here been presented to DSolve as a list.

Much of our manipulation in the above illustration would have been simplified if we had expressed the original equation in dimensionless form before seeking its solution. To that end, we submit the statements

\[
\text{In}[21]:= \text{Quit[]}
\]

\[
\text{In}[1]:= \text{deq1} = m*x''[t] + b*x'[t]+k*x[t]==0
\]

\[
\text{Out}[1]= kx[t] + bx'[t] + mx''[t] == 0
\]

to clear the workspace and redefine the equation. Then, with the statements

\[
\text{In}[2]:= \text{deq2} = \text{deq1} /. \{k -> m*[\text{Omega}]^2, b -> 2*m*[\text{Gamma}]\}
\]

\[
\text{Out}[2]= m\omega^2 x[t] + 2m\gamma x'[t] + mx''[t] == 0
\]

\[
\text{In}[3]:= \text{deq3} = \text{Cancel}[\text{deq2[[1]]}/m ] == 0
\]

\[
\text{Out}[3]= \omega^2 x[t] + 2\gamma x'[t] + x''[t] == 0
\]

we introduce the quantities \( \omega \) and \( \gamma \) defined by \( \omega^2 = k/m \) and \( \gamma = b/(2m) \), where \( \omega \) is the natural frequency of the oscillator, and remove the parameter \( m \) from the picture altogether. Next, we change the independent variable to \( \tau = \omega t \). One successful process for achieving that end involves

(1) introducing a new function with the statement

\[
\text{In}[4]:= x[t_] := y[\text{\Omega}]*t
\]

(2) reevaluating \text{deq3} with the statement

\[
\text{In}[5]:= \text{deq4} = \text{deq3}
\]

\[
\text{Out}[5]= \omega^2 y[t \omega] + 2\gamma y'[t \omega] + \omega^2 y''[t \omega] == 0
\]

(3) replacing the quantity \( \omega t \) (which is now the variable with respect to which differentiation is to be evaluated in \text{deq4}) with a single variable \( \tau \) by executing the statement
8.8. USING MATHEMATICA...  

8.8.10 ... for Vector Calculus

*Mathematica* includes several commands to facilitate vector calculus. The main commands are Grad, Div, Curl, and Laplacian for the gradient, divergence, curl, and Laplacian, respectively. Each accepts a first argument—a scalar function for Grad and Laplacian; a vector function for Div and Curl—and a second argument, a list that identifies the variables in the coordinate system in which the quantity is expressed. An optional third argument—a string—specifies the coordinate system to be used, with "Cylindrical" and "Spherical" as the main alternatives to the default ("Cartesian").

---

32 Here, the assumption $\beta < 1$ is equivalent to the assumption $\gamma/\omega < 1$ or $\gamma^2 < \omega^2$ or $b^2/4m^2 < k/m$ or $b^2 < 4mk$. Thus, the *dimensionless* inequality $\beta < 1$ here is exactly equivalent to the *dimensional* inequality $b^2 < 4km$ that appeared earlier in this section.

33 Prior to Version 9, vector calculus was provided in Mathematica by the add-on package VectorAnalysis, which needed to be added with the statement `<< VectorAnalysis`. In subsequent versions, this capacity was made part of the main kernel, though with rather different commands and syntax. The previous package may remain available for a while, but it is officially obsolete and is likely to disappear altogether at some point.
We begin this illustration by defining scalar and vector quantities in each of the three common coordinate systems with the statements

\begin{verbatim}
In[1]:= scar = x^2+y^2+z^2
Out[1]= x^2 + y^2 + z^2

In[2]:= scyl = r^2+z^2
Out[2]= r^2 + z^2

In[3]:= ssph = r^2
Out[3]= r^2

In[4]:= vcar = {x,y,z}
Out[4]= {x,y,z}

In[5]:= vcyl = {r,0,z}
Out[5]= {r,0,z}

In[6]:= vsph = {r,0,0}
Out[6]= {r,0,0}
\end{verbatim}

Define a scalar in Cartesian coordinates.
Define a vector in Cartesian coordinates.

Define a scalar in cylindrical coordinates.
Define a vector in cylindrical coordinates.

Define a scalar in spherical coordinates.
Define a vector in spherical coordinates.

Then, we evaluate the gradients with the statements,

\begin{verbatim}
In[7]:= Grad[scar, {x,y,z}]
Out[7]= {2x, 2y, 2z}

In[8]:= Grad[scyl, {r,\[Phi],z}, "Cylindrical"]
Out[8]= {2r, 0, z}

In[9]:= Grad[ssph, {r,\[Theta],\[Phi]}, "Spherical"]
Out[9]= {2r, 0, 0}
\end{verbatim}

Evaluate \( \nabla \) in Cartesian coordinates.
Evaluate \( \nabla \) in cylindrical coordinates.
Evaluate \( \nabla \) in spherical coordinates.

the Laplacian with the statements

\begin{verbatim}
In[10]:= Laplacian[scar, {x,y,z}]
Out[10]= 6

In[11]:= Laplacian[scyl, {r,\[Phi],z}, "Cylindrical"]

In[12]:= Laplacian[ssph, {r,\[Theta],\[Phi]}, "Spherical"]
Out[12]= 6
\end{verbatim}

Evaluate \( \nabla^2 \) in Cartesian coordinates.
Evaluate \( \nabla^2 \) in cylindrical coordinates.
Evaluate \( \nabla^2 \) in spherical coordinates.

the divergence with the statements

\begin{verbatim}
In[13]:= Div[vcar, {x,y,z}]
Out[13]= 3

In[14]:= Div[vcyl, {r,\[Phi],z}, "Cylindrical"]
Out[14]= 3

In[15]:= Div[vsph, {r,\[Theta],\[Phi]}, "Spherical"]
Out[15]= 3
\end{verbatim}

Evaluate \( \nabla \cdot \) in Cartesian coordinates.
Evaluate \( \nabla \cdot \) in cylindrical coordinates.
Evaluate \( \nabla \cdot \) in spherical coordinates.

and, finally, the curl with the statements

\begin{verbatim}
In[16]:= Curl[vcar, {x,y,z}]
Out[16]= {0,0,0}

In[17]:= Curl[vcyl, {r,\[Phi],z}, "Cylindrical"]
Out[17]= {0,0,0}

In[18]:= Curl[vsph, {r,\[Theta],\[Phi]}, "Spherical"]
Out[18]= {0,0,0}
\end{verbatim}

Evaluate \( \nabla \times \) in Cartesian coordinates.
Evaluate \( \nabla \times \) in cylindrical coordinates.
Evaluate \( \nabla \times \) in spherical coordinates.

*Mathematica* can also evaluate these vector derivatives when the dependence in the first argument on the coordinates is not explicit. For example

\begin{verbatim}
In[19]:= Laplacian[ f[r,\[Phi],z], {r, \[Phi], z}, "Cylindrical" ]
Out[19]= f^{(0,0,2)}[r,\phi,z] + f^{(0,2,0)}[r,\phi,z]/r + f^{(1,0,0)}[r,\phi,z] + f^{(2,0,0)}[r,\phi,z]
\end{verbatim}
8.9 Packages

Here, the superscripts on \( f \) indicate how many derivatives with respect to each of the three variables are to be evaluated, e.g.,

\[
f^{(2,0,0)}[r, \phi, z] = \frac{\partial^2 f}{\partial r^2} \quad ; \quad f^{(1,1,0)}[r, \phi, z] = \frac{\partial^2 f}{\partial r \partial \phi}
\]

where 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 Grad, Div, Laplacian, and Curl for working with coordinate dependent vectors and scalars. Mathematica also includes a variety of commands to transform the coordinates of a point from one system to another, to calculate the area or volume of appropriately defined regions, to create two and three dimensional graphical displays of vector fields, and many more. You should make it a point to bring up Wolfram Documentation from the NBI HELP menu, search for ‘Vector Analysis’ in that documentation, and study the multitude of available commands. Be aware, also, that Mathematica can apply these commands to functions and vectors in more than three dimensions.

8.9 Packages

To this point, we have supposed that the Mathematica we are using has been configured in the way set up by the default installation, so that all of the functions so far invoked will be available to us once Mathematica has been launched. At some sites, however, Mathematica may have been configured to conserve memory, and some of the features normally loaded automatically may have been left out. At other sites, additional features giving further commands may have been included when Mathematica is started. This flexibility is provided through the use of Mathematica packages, segments of Mathematica code which may or may not be automatically loaded when Mathematica is started up and which define features that will be present only when the package defining those features has been loaded. Any package not loaded automatically can, of course, be explicitly loaded to make its features available. A list of the packages included with the standard Mathematica distribution can be found by accessing the Wolfram Documentation either from the HELP menu in the NBI or on line, searching for ‘Packages’, and opening the link labeled something like ‘Wolfram System Standard Extra Packages’. The resulting page has links to descriptions of the packages. A list of the packages that have been loaded into the current Mathematica configuration is contained in the special variable \$Packages, whose value can be displayed by the simple statement.\[34\]

\begin{verbatim}
In[9]:= $Packages
Out[9]= ListOfLoadedPackages
\end{verbatim}

If a needed package is not loaded automatically when Mathematica is launched, it can be explicitly loaded with one of the commands.\[35\]

\begin{verbatim}
In[23]:= <<PackageName'
In[38]:= Get['PackageName']
In[79]:= Needs['PackageName']
\end{verbatim}

Here << and Get are synonymous and will load the package directly, even if it has previously been loaded. In contrast, Needs will load the package only if it has not already been loaded.

Beyond the standard packages, individual users can create additional packages for special purposes. Thus, if we are lucky, we might be able to find a user-contributed package with special features

\[34\] The list output by this statement may vary from site to site, depending on how Mathematica is configured. It will also reflect the inclusion of any packages explicitly loaded during the session.

\[35\] Note that the tic following PackageName in the first statement is a forward tic.
of value to a particular exploration. In the absence of such luck, we could also create our own package to serve those purposes. The details of writing such packages are laid out in the Mathematica manuals but are beyond the scope of this book.

With each new version of Mathematica, some features that had been provided with add-on packages may be moved to the main kernel, in which case those features will be available without explicitly loading the package. Indeed, prior to Version 9, the capacity for evaluating several vector derivatives described in Section 8.8.10 was available only if a package named VectorAnalysis was loaded explicitly. Starting with Version 9, those features were built into the main kernel, though the names of the commands and their syntax suffered revision. For backward compatibility, the original package is retained in subsequent versions, at least for a while, but it may in time be completely removed. Further, loading the package will also generate a message alerting the user that the package has become obsolete.

8.10 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 Mathematica. The simplest is the For loop, which we have already met in Sections 8.8.2 and 8.8.5. Briefly, the statements

\begin{verbatim}
In[1]:= x = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 } 
Out[1]= {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
In[2]:= For[ i = 1, i <= 10, i = i + 2, x[[i]] = i^2 ] 
In[3]:= x
Out[3]= {0, 0, 9, 0, 25, 0, 49, 0, 81, 0}
\end{verbatim}

create a 10-element list of zeros and then assign the squares of the odd integers to the odd elements of that list. Note that the C-like structure \texttt{i++} can be used instead of the expression \texttt{i = i + 1} if the index is to be incremented by 1.

In essence, the For loop requires \textit{a priori} knowledge of how many times the loop will be executed. For cases when that knowledge is not in hand, Mathematica makes available the While loop, whose general syntax has the form

\begin{verbatim}
While[ ⟨condition⟩, ⟨block of statements⟩ ]
\end{verbatim}

In this loop, the block of instructions must assume responsibility for changing the condition to a value that will stop the loop. More specifically, these loops are illustrated in the statements

\begin{verbatim}
In[4]:= x = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
In[5]:= i = 1;
In[6]:= While[ i < 11, x[[i]] = i^3; i = i+2 ];
In[7]:= x
Out[7]= {0, 0, 0, 27, 0, 125, 0, 343, 0, 729, 0}
\end{verbatim}

Note that,

- The block of statements in this structure starts after the first comma and continues until the closing bracket.
- There is \textit{no} punctuation after the last statement in the block of statements.
- Individual statements in that block are separated by semicolons.
• The statement incrementing \(i\) is essential so that the initially true condition \(i < 11\) is ultimately toggled to false and the loop terminates.

Finally, a word about composite conditions: The illustrations above made use of (single) conditions like \(i < 11\), which will evaluate to **true** or **false** depending on the value of \(i\). Indeed, the *Mathematica* function *Boole*, for example,

\[
\begin{align*}
\text{In[8]:=} & \quad i := 5; \\
\text{In[9]:=} & \quad \{ \text{Boole}[i > 10], \text{Boole}[i < 10] \} \\
\text{Out[9]=} & \quad [0, 1]
\end{align*}
\]

allows the testing of a logical condition. In this Boolean context, 0 equates to false and 1 to true. The six logical operators \(==, <, >, <=, >=, \text{ and } !=\) 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. *Mathematica* makes available the logical operators \(&&\), \(||\), and \(!\), and the logical functions \(\text{Xor}\) (exclusive or), \(\text{Nand}\), and \(\text{Nor}\) to facilitate combining conditions and negating conditions. More specifically, with \(\langle C_1 \rangle\) and \(\langle C_2 \rangle\) individual conditions

- \((\langle C_1 \rangle \&\& \langle C_2 \rangle)\) will be **true** if \(\langle C_1 \rangle\) and \(\langle C_2 \rangle\) are both true and **false** otherwise,
- \((\langle C_1 \rangle || \langle C_2 \rangle)\) will be **true** if either \(\langle C_1 \rangle\) or \(\langle C_2 \rangle\) is true or both \(\langle C_1 \rangle\) and \(\langle C_2 \rangle\) are true and **false** otherwise, and
- \(\text{Xor}[\langle C_1 \rangle, \langle C_2 \rangle]\) will be **true** if either \(\langle C_1 \rangle\) or \(\langle C_2 \rangle\) is true and **false** if neither condition is or both are true, and
- \(!\langle C_1 \rangle\) will be **true** if \(\langle C_1 \rangle\) is false and **false** if \(\langle C_1 \rangle\) is true.

For example, the coding

\[
\begin{align*}
\text{In[10]:=} & \quad x = \{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 \}; \\
\text{In[11]:=} & \quad i = 1; \\
\text{In[12]:=} & \quad \text{While}[ (i > 0 \&\& \ i < 11), x[[i]] = i^3; \ i = i + 1 ] \\
\text{In[13]:=} & \quad x \\
\text{Out[13]=} & \quad [1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
\end{align*}
\]

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, *Mathematica* also provides the function \(\text{If}\) for this purpose. Its fundamental syntax is

\[
\text{If[ } \langle \text{condition} \rangle, \langle \text{statements if true} \rangle, \langle \text{statements if false} \rangle \]
\]

though the second block of statements can be omitted if nothing is to be done when the condition is false. Thus, for example, the statement

\[
\text{For[ } i=1, i<=10, i++, \text{If[ } x[[i]] < 0, x[[i]] = -x[[i]] \] ]}
\]

will replace each negative element in a list \(x\) with the corresponding positive value, the statement

\[
\text{For[ } i = 1, i<=10, i++, \text{If[ } x[[i]] > 10, x[[i]] = 10 \] ]}
\]
will replace all values greater than 10.0 with the value 10.0, and the statement

\[
\text{If}[\text{a} \geq 0, \text{b} = \text{a}, \text{b} = -\text{a}]
\]

will set \( \text{b} \) equal to the absolute value of \( \text{a} \) (though the function \text{Abs} will do so more easily). Multiple statements in either the true clause or the false clause will be separated by commas and enclosed within braces, as in

\[
\text{If}[\langle \text{condition} \rangle, \{ \langle \text{statement} \rangle, \langle \text{statement} \rangle, \ldots \}]
\]

For selection among more than two options, the statement would have to be nested with care. 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.

### 8.11 Command Files

In addition to responding to statements supplied interactively, \text{Mathematica} can execute statements read from a command file. To begin, we create a text file containing the desired statements just as they would be typed interactively. If that file is then stored in the default directory\(^{36}\) or somewhere in \text{Mathematica}'s search path,\(^{37}\) execution of the command(s) contained within it would then be specified by typing either of the statements

\[
\text{<< File\_Name} \quad \text{or} \quad \text{Get["File\_Name"]}
\]

when \text{Mathematica} is ready to accept input. There is no default file type.\(^{38}\) As the command file is read and executed, the statements themselves will \textit{not} be displayed on the screen. Whether the statements are terminated with semicolons or not, the output will also not be displayed on the screen (except for the last statement in the file, whose output will be displayed if it ends with no semicolon and suppressed if it ends with a semicolon). Further, no line labels will be attached to any of the intermediate results, and the output from the last statement in the file (if any) will be given the label Out\([n]\), where \( n \) is the identifying number of the statement requesting the reading of the file. Only if intermediate results are explicitly printed (with the command \text{Print[]} \) will these results be displayed; only if intermediate results are explicitly bound to a variable will those results be available once execution of the command file has been completed. Except for the absence of the In and Out labels, the \text{Mathematica} environment after the command file has been executed 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 \text{Mathematica}. 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 \text{Mathematica}; repeated typing of the correct parts of the sequence is not then necessary. To illustrate matrix multiplication, for example, we could create the file \text{testmultiply.m} containing the statements listed in Table 8.1.\(^{39}\)

Then, assuming that this file has been stored in the default directory,\(^{40}\) we would execute the statements it contains by submitting to \text{Mathematica} one or the other of the statements

\(^{36}\)See Section 8.18.3.

\(^{37}\)See Section 8.18.4.

\(^{38}\)Conventionally, text files containing \text{Mathematica} statements are given names with file type \text{.m}. Further, the types \text{.nb} and \text{.mx} are conventionally used for files containing \text{Mathematica} notebooks and \text{Mathematica} definitions in a special format, respectively.

\(^{39}\)Comments in \text{Mathematica} code are introduced with the symbol (* and terminated with the symbol *).

\(^{40}\)See again Section 8.18.3.
8.11. COMMAND FILES

Table 8.1: The Mathematica command file testmultiply.m.

(* Command file testmultiply.m *)

(* Enter and display matrix *)
mat1 = { {1,2}, {3,4} }
Print[ MatrixForm[ mat1 ] ]

(* Evaluate and display transpose *)
mat2 = Transpose[ mat1 ]
Print[ MatrixForm[ mat2 ] ]

(* Evaluate and display matrix product *)
Print[ MatrixForm[ mat1.mat2 ] ]

(* Evaluate and display element-by-element product *)
Print[ MatrixForm[ mat1*mat2 ] ]

Table 8.2: The Mathematica command file crossdot.m.

(* crossdot.m *)

(* Command file to define functions to evaluate the cross and dot products of two three-component vectors. *)
lucross[a_,b_] :=
{ a[[2]]*b[[3]] - a[[3]]*b[[2]],
a[[3]]*b[[1]] - a[[1]]*b[[3]],
a[[1]]*b[[2]] - a[[2]]*b[[1]] };
ludot[a_,b_] := a[[1]]*b[[1]] + a[[2]]*b[[2]] + a[[3]]*b[[3]];

<< testmultiply.m or Get[ "testmultiply.m" ]

All output produced by the Print statements will be displayed on the screen but only the expressions explicitly bound to mat1 and mat2 will be available by those names once execution of the file is completed.

As a second example, suppose 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 listed in Table 8.2. Here, beyond the comments, the first four lines—actually a single statement—define the function lucross, 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 ludot, which takes the same arguments but returns a scalar value equal to the dot product of those two lists treated as three-dimensional vectors. This file is named crossdot.m and can be accessed in the directory $HEAD/mathematica. Thus, for example, the statement

41 See the note at the beginning of this chapter.
CHAPTER 8. INTRODUCTION TO MATHEMATICA

Get[ "$HEAD/mathematica/crossdot.m" ];

will load this file from the directory associated with this book, defining the two functions ludot and lucross for subsequent use.\footnote{Be aware, as well, that the dot product of two vectors can also be evaluated with the non-commutative infix operator ., which will actually provide the sum of the products of corresponding components regardless of the dimensionality of the vectors. Further, in addition to the several vector derivatives already introduced in Section 8.8.10, Mathematica includes the commands Dot and Cross, which are more carefully structured than ludot and lucross to evaluate dot and cross products.}

Finally in this section, we note that Mathematica also provides means by which we can display the contents of a command file conveniently and easily from within Mathematica. To display the file crossdot.m, we submit to Mathematica the statement

FilePrint[ "crossdot.m" ] or FilePrint[ "$HEAD/mathematica/crossdot.m" ]

Here, the first form applies if the file is in a directory in the search path and the second form refers to the file in the directory associated with this book. The contents of the file will be displayed in the NBI or, if you are using the TBI, in the Shell window from which Mathematica was launched.

8.12 High-Resolution Graphing

We illustrate the production of high resolution graphics by plotting the solution for the damped harmonic oscillator obtained in Section 8.8.9. First, we reclaim the solution with the statement

\begin{verbatim}
In[1]:= soln3 = x0 * Exp[-\[Gamma]*t] * (\[Gamma]*Sin[a*t]/a + Cos[a*t])
Out[1]=
e^{-t\gamma}x_0\left(\frac{\gamma\sin[at]}{a} + \cos[at]\right)
\end{verbatim}

In addition, we calculate and display the velocity with the statement

\begin{verbatim}
In[2]:= soln3dot = Simplify[ D[ soln3, t ] ]
Out[2]=
\frac{-e^{-t\gamma}x_0(a^2 + \gamma^2)\sin[at]}{a}
\end{verbatim}

Then, recognizing that \( \gamma \) and \( a \) are not independent \((a^2 = k/m - \gamma^2)\) and hence that assigning values to \( \gamma \) and \( a \) in effect fixes \( k/m \), we use the statements

\begin{verbatim}
In[3] := \[Gamma] = 0.1; a = 1.0; x0 = 1.0;
\end{verbatim}

\begin{verbatim}
In[4]:= Plot[ {soln3,soln3dot}, {t,0.0,20.0} ]
\end{verbatim}

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 Mathematica TBI and as a smooth plot in the NBI.\footnote{The number of points distributed over the range of the independent variable is determined in Mathematica to keep the amount of turning from segment to segment small. Numerous options—see the Mathematica manuals—give the user the ability to override the defaults.}

Alternatively, we could modify the resulting graph by exploiting a few of the many options supported by the command Plot. The statement

\begin{verbatim}

```mathematica
In[1]:= soln3 = x0 * Exp[-\[Gamma]*t] * (\[Gamma]*Sin[a*t]/a + Cos[a*t])
Out[1]=
e^{-t\gamma}x_0\left(\frac{\gamma\sin[at]}{a} + \cos[at]\right)

In[2]:= soln3dot = Simplify[ D[ soln3, t ] ]
Out[2]=
\frac{-e^{-t\gamma}x_0(a^2 + \gamma^2)\sin[at]}{a}

In[3] := \[Gamma] = 0.1; a = 1.0; x0 = 1.0;

In[4]:= Plot[ {soln3,soln3dot}, {t,0.0,20.0} ]
```

```mathematica
\end{verbatim}

```mathematica
```

```mathematica
```

```mathematica
\end{verbatim}

```mathematica
```

```mathematica
```

```mathematica
\end{verbatim}

```mathematica
```

```mathematica
```

```mathematica
```
Figure 8.1: Graph of position and velocity for a damped oscillator with $\gamma = 0.1$, $a = 1.0$, $x(0) = x_0 = 1.0$, and $v(0) = 0.0$. Here, the solid and dashed lines show $x(t)$ and $v(t)$, respectively.

\[ x,v \]

\[ \gamma=0.1, a=0.1 \]

```
In[5]:= Plot[ {soln3, soln3dot}, {t,0.0,20.0}, PlotRange->{-1.0,1.0}, PlotPoints->100, AxesLabel->"t","x,v"],
PlotLabel->StyleForm[ "Damped Oscillator", 
FontSize->16, FontFamily->"Times", FontWeight->"Bold" ],
PlotStyle->{ {Thickness[0.01],Black},
{Thickness[0.01],Black,Dashing[{0.02,0.02}] } },
Epilog->{Text["$\gamma=0.1, a=0.1$", {15.0, 0.75}] } ]
```

for example, will produce the same graph but with

- the vertical axis running from $-1.0$ to $+1.0$,
- at least 100 points spaced over the range of the independent variable (the default is 25),
- the indicated labels on the horizontal and vertical axes,
- the indicated title in 16-point, bold type, Times font on the graph as a whole,
- the first expression (but not the axes) displayed black with a line of thickness heavier than the default (i.e., with a thickness set to the indicated fraction of the height of the graph as a whole) and the default (solid) linestyle and the second expression displayed black with the same line thickness but with a dashed line style with alternate dashes and gaps, each of size 0.02 units, and
- the text $\gamma = 0.1, a = 0.1$ displayed on the face of the graph and centered at the point [15.0, 0.75].

These (and other) options to the command Plot are fully described in the Mathematica manuals. The graph resulting from the statement on line In[5] is shown in Fig. 8.1.

We could explore other features of this motion by, for example, changing the value of $\gamma$ and reploting the graph with the statements

\[ 44\text{See the Mathematica manuals at Epilog and Text for details on the way to specify text displayed on a graph.} \]
Figure 8.2: Graph of velocity versus position for a damped oscillator with $\gamma = 0.2$, $a = 1.0$, $x(0) = 1.0$, and $v(0) = 0.0$.

In[6]:= $\gamma = 0.2$;
In[7]:= Block[ {soln3, soln3dot}, {t, 0.0, 20.0},
PlotRange -> {-1.0, 1.0},
PlotPoints -> 100, AxesLabel -> {"t", "x,v"},
PlotLabel -> StyleForm["Damped Oscillator",
FontSize -> 16, FontFamily -> "Times", FontWeight -> "Bold"],
PlotStyle -> {Thickness[0.01], Black},
{Thickness[0.01], Black, Dashing[0.02, 0.02]} ]

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

In[8]:= ParametricPlot[{soln3, soln3dot}, {t, 0.0, 20.0},
PlotRange -> {-1.0, 1.0}, {-1.0, 1.0},
PlotStyle -> {Thickness[0.01], Black}, AspectRatio -> 1.0 ]

Here, the new option AspectRatio permits us to tell Mathematica to use a plotting area that is square (aspect ratio 1.0). The resulting graph is shown in Fig. 8.2.

Mathematica can, of course, also produce surface plots and contour plots in several different ways. The simplest and quickest such plots are produced with statements like

In[9]:= mode23 = Sin[ 2*Pi*x ] * Sin[ 3*Pi*y ]
In[10]:= Plot3D[ mode23, {x, 0.0, 1.0}, {y, 0.0, 1.0},
PlotPoints -> 25, ColorFunction -> GrayLevel ]
In[11]:= lvls = Range[ -9, 9, 1 ]/10.0;
In[12]:= ContourPlot[ mode23, {x, 0.0, 1.0}, {y, 0.0, 1.0},
AspectRatio -> 1.0, ContourShading -> False,
Contours -> lvls, PlotPoints -> 50 ]
In[13]:= Quit[]
Figure 8.3: Surface plot of the function $z(x, y) = \sin(2\pi x) \sin(3\pi y)$.

Line \texttt{In[9]} here defines the expression to be displayed. Line \texttt{In[10]} produces a surface plot with 25 points (default 15) along each axis and coloring set to a gray scale (since we anticipate black and white printing in this book). Finally, line \texttt{In[11]} defines the values at which contour lines are too be drawn (from $-0.9$ to $0.9$ in steps of $0.1$) and line \texttt{In[12]} produces a contour map at those levels (option \texttt{Contours}) with shading turned off, the aspect ratio adjusted so that the display will be square, and the function evaluated at 50 points along each axis. The resulting graphs are shown in Figs. 8.3 and 8.4.

We have, of course, only mentioned a few (Plot, ParametricPlot, Plot3D, and ContourPlot) of the functions and only some (PlotRange, PlotPoints, AxesLabel, PlotLabel, PlotStyle, FontSize, FontWeight, ColorFunction, AspectRatio, ContourShading, and Contours) of the options that modify the behavior of these functions. Several other functions for producing graphical displays and numerous additional options are available. You should make it a point to read about these functions and the associated options in the \textit{Mathematica} manuals. In particular, note that the statement \texttt{Options[\textit{Function}]\texttt{}}\textsuperscript{45} will return a list of all of the options that affect \textit{Function}, attaching to each its current default value.

\section*{8.13 Making Hard Copy . . .}

The method used to make hard copy of all or part of a \textit{Mathematica} dialog depends on whether the TBI or the NBI 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.

\textsuperscript{45}See Section 8.18.5 for a fuller discussion.
8.13.1 ... from the NBI

The menu structure and procedures to print text and graphics from the NBI may differ from version to version and platform to platform. For obtaining hard copy in the NBI,\footnote{This description has been worked out in LINUX and Windows environments. The same features will be available in other environments, but the details of the behavior may be slightly different. Further, the details here described apply specifically to Mathematica 11. Other versions may make these capabilities available on other ways.} several items in the File menu are relevant:

- The option ‘Print...’ opens a pop-up window in which to specify a printer\footnote{‘Print to file’ is among the options but offers fewer formats than ‘Save’, ‘Save As’, and ‘Save Selection As’} and, if the OPTIONS or PREFERENCES menu is opened, select various parameters (page range to be printed, number of copies to be printed, treatment of color in the output, …). In Windows, a capacity to print a selection from the active notebook is provided as an option in the ‘Page Range’ panel of the ‘Print...’ option in the File menu; in UNIX, a capacity to print a selection from the active notebook is provided in the separate option ‘Print Selection...’ in the File menu. In either case, the option will be available only if a portion of the notebook has, in fact, been selected.

- The option ‘Printing Settings→’ from the File menu will open a menu offering several options, including
  
  - ‘Page Setup’, which opens a pop-up window in which to specify paper size, orientation (portrait or landscape), and margins.
  - ‘Printing Options’, which opens a pop-up window in which to specify whether to print cell brackets, registration marks, and a few other features.
  - ‘Headers and Footers’, which opens a pop-up window in which to specify the content and position of desired page headers and footers.
  - ‘Show Page Breaks’, a check box, to select whether page breaks should be displayed or hidden in the NBI.
  - ‘Printing Environment’, which opens a pop-up window in which to select among the options ‘Working’, ‘Slide Show’, or ‘Printout’.

}\footnote{This description has been worked out in LINUX and Windows environments. The same features will be available in other environments, but the details of the behavior may be slightly different. Further, the details here described apply specifically to Mathematica 11. Other versions may make these capabilities available on other ways.}
8.13. MAKING HARD COPY ...

- The option 'Print Preview...' displays on screen what will ultimately be printed. That on-screen display will be produced in an Adobe Reader window from which a PDF file can, of course, be saved.

- The options 'Save...', 'Save As...', and 'Save Selection As...', each of which brings up a browser in which you can identify the directory in which all or the selected part of a notebook should be saved, the name to give the resulting file, and the file type to be used. Available file types include tex, ps, eps, pdf, and html, though not all types are available with all options. In particular, the filetype ps is available with 'Save...' and 'Save As...' and the file type eps is available with 'Save Selection As'. The resulting file can then be printed directly or incorporated in a more extensive document.

These routes to output information include the Mathematica-generated labels for all included elements of the notebook.

Mathematica also provides a facility for writing a description of a specified graphic object (not the entire notebook) into a file. We simply need to inform Mathematica 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, with graph the label of the graphics display to be output to a file, either of the statements

Export[ "trial.eps", graph, "EPS" ]

which determines the format of the output file from the third argument, or

Export[ "trial.eps", graph ]

which determines the format of the output file from the extension in the file name, will tell Mathematica to convert the graphic object identified by the variable graph into encapsulated PostScript and write that description to the file trial.eps in the default directory. When displayed on a computer screen with ghostview, an EPS file creates a display that includes a number of controlling buttons surrounding the graph itself. These surrounding buttons are not included in PDF or JPEG files, nor do they appear, for example, when the EPS file is incorporated in a \LaTeX{} document. This route to exporting a graph does not include Mathematica-generated output labels in the file.

Note, however, that in some contexts Mathematica may not always transcribe all characters in the text in the on-screen graphic (e.g., in labels on the axes or in titles) into the proper PostScript characters. Unless some Mathematica-specific fonts have been made available to the program that will interpret the PostScript file, opening and closing parentheses, the forward slash, and probably other characters may be rendered in unexpected forms when the resulting PostScript file is printed or viewed on the screen. In exporting graphics, we must either be sure that the appropriate Mathematica fonts are known by the program(s) that will display the file, or we must constrain Mathematica as it produces the graphic by limiting its choice of fonts for use in the graphic. The former route is complicated (and necessary if, for example, Greek letters are to be included in the exported text). While not always satisfactory, the latter route is far simpler; it involves the stipulation of an additional option in the command producing the graphic. For the sake of a quick illustration, we simply note that the statements

\begin{verbatim}
graph = Plot[ Sin[t], {t, 0, 10}, AxesLabel->{"sin(t)", "x/a"},
    FormatType->OutputForm ];
Export[ "trial.eps", graph, "EPS" ];
\end{verbatim}

will produce an encapsulated PostScript file in which the parentheses and the slash in the labels on the axes will be properly rendered, even when the specific Mathematica fonts are not available to the program that will display trial.eps.

---

48 Other available formats include "PDF" (file type pdf) and "JPEG" (file type jpeg). A full list is stored in the system variable \$ExportFormats.

49 See Section 8.18.3.

50 For details, see the Mathematica manuals.
CHAPTER 8. INTRODUCTION TO MATHEMATICA

8.13.2 ... from the TBI

Printing of text displayed in the TBI is most simply accomplished by using the cutting and pasting capabilities of the operating system to transfer that character information into a text file, which can then be edited in a standard text editor and sent to an available printer with the standard commands to the operating system (or perhaps by selecting appropriate options from a menu available with the editor).

Graphics commands like Plot may not produce very satisfactory on-screen results when executed in the TBI. Even so, if it is given a variable name, a graphic object can still be written into a file as high-resolution graphic for subsequent printing. As in the previous section, we simply need to inform Mathematica 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, either of the statements

\begin{verbatim}
Export[ "trial.eps", graph, "EPS" ]
\end{verbatim}

or

\begin{verbatim}
Export[ "trial.eps", graph ]
\end{verbatim}

will tell Mathematica to convert the graphic object identified by the variable graph into encapsulated PostScript and write that description to the file trial.eps in the default directory.

8.14 Output in \LaTeX Format

A particularly useful feature of Mathematica is its ability to write \LaTeX versions of expressions to the screen or into a file. Full details are laid out in the Mathematica manuals. In brief, the statement\footnote{Note the upper-case X in the third character.}

\begin{verbatim}
TeXForm[ expr1 ]
\end{verbatim}

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, either of the statements

\begin{verbatim}
TeXForm[ expr1 ] >> FileName
Put[ TeXForm[ expr1 ], "FileName" ]
\end{verbatim}

will create a new (ASCII) file and write the \LaTeX description of the indicated expression into that file, overwriting any existing file by the same name. To append the \LaTeX description of the expression to an existing file, one or the other of the statements

\begin{verbatim}
TeXForm[ expr1 ] >>> FileName
PutAppend[ TeXForm[ expr1 ], "FileName" ]
\end{verbatim}

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 Mathematica 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 formatting instructions will have to be supplied through manual editing of the file created by Mathematica. Further, even more subtle edits may be necessary.\footnote{52}{Conveniently, variables named \texttt{$\Gamma$} and \texttt{\CapitalGamma}, for example, will be properly translated into \gamma and \Gamma.}

Even more generally, if we have created a Mathematica notebook in the NBI, we can output that notebook in its entirety (including the textual sections) into a \LaTeX\ file by

- Selecting ‘Save as . . .’ from the File menu,
- Selecting the \LaTeX\ option in the text entry box labeled ‘Files of type:’,
- Specifying the desired file name in the text entry box labeled ‘File name:’,
- Browsing to the desired directory, and
- Clicking ML on ‘Save’.

The file will be written into the specified directory. The resulting file is a complete \LaTeX\ source file with all necessary packages and new command definitions in its preamble. If the content of this file is to be incorporated into another \LaTeX\ source file, some of the commands at its beginning and end will have to be stripped off but any \texttt{\usepackage} and \texttt{\newcommand} commands not already in the destination file must be copied from the Mathematica-produced file.\footnote{53}{For example, output of the Mathematica expression \texttt{Sin[x+y]/Cos[x-y]} with \texttt{TeXForm} yielded the correct but unintended result \texttt{$\sin (x+y) \sec (x-y)$}. Note also the unnecessary spaces after \texttt{sin} and \texttt{cos}.}

8.15 Animation

The Mathematica command \texttt{Animate} facilitates the creation of animated displays in the NBI. The simple statement

\begin{verbatim}
Animate[ Plot[Sin[Pi*x]*Cos[2*Pi*t], {x,0.0,1.0},
               PlotRange->{ {0.0,1.0},{-1.0,1.0} }], {t, 0.0, 1.0, 0.05} ]
\end{verbatim}

will, for example, display the fundamental mode of a vibrating string, keeping the axes the same size throughout all frames. A more complicated animation, display of the 23 mode of a square membrane, is produced by the statement

\begin{verbatim}
Animate[ Plot3D[Sin[2*Pi*x]*Sin[3*Pi*y]*Cos[2*Pi*t], {x,0.0,1.0}, {y,0.0,1.0},
                   PlotRange-> { {0.0,1.0},{0.0,1.0},{-1.0,1.0}}], {t,0.0,1.0,0.05},
                  AnimationRate-> 0.1]
\end{verbatim}

In both cases, the button labeled ‘Pause’ in the window displaying the animation will stop the animation and allow subsequent statements to be submitted to Mathematica.

Unfortunately, the command \texttt{Animate} has too many possibilities and options for full discussion here. Details can be found in the Mathematica manuals.
8.16 Using the Notebook

Within the NBI, the window in which Mathematica 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 Mathematica commands. Notebooks are constructed out of cells, which may contain executable Mathematica statements (and the associated output)—the only cells we have so far seen—or may contain text. For purposes of organizing larger documents, these cells 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 Mathematica notebook containing a few Mathematica statements and their output. Note, in particular, that each statement and its output embraces two cells, one containing the input statement and the other containing the associated Mathematica response. The input cell is marked at the right side of the Notebook window with a closing square bracket whose upper corner contains also a diagonal line, and the output cell is similarly marked except that the bracket also has a short horizontal line below the diagonal line. Further, the two cells are themselves grouped together at the right side of the Notebook window with a closing square bracket that has no adornments at its corners. Finally, note that double clicking ML on the outer square bracket will “close” the pair, hiding the output and leaving only the input statement. In this arrangement, the outer bracket now has half an arrowhead at its lower end to indicate that the cell has been closed. Double clicking ML on the outer bracket when it is in this state will reopen the cell.

Now, suppose that we want to precede each statement with an explanatory paragraph. We could achieve that end for one of the statements by

1. Moving the cursor to the position at which a new cell is desired, placing it so that it becomes a horizontal I-beam.
2. Clicking ML to create a horizontal line (cell insertion bar) across the screen.
4. Typing the desired text, using the various options in the Format menu to specify font, type size, type style, alignment, ... As you start typing, a new cell will be created for the typed text. This cell will be bracketed on the right with a square bracket that has a short horizontal line below its top bar—the identifier of a text cell.

This process can, of course, be repeated to provide a textual description for each of the remaining statements.

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 simply notice that the Style submenu of the Format menu contains numerous items beyond the one introduced in the previous paragraph. Positioning the cursor at the point of a desired textual insertion, clicking ML to create a horizontal line (cell insertion bar) across the screen, and then selecting some other item (Title, Subtitle, Section, Subsection, ...) from the Style submenu of the Format menu will then set Mathematica to create a cell of the selected style when we begin typing. Further, Mathematica will select a type size and weight appropriate to the selected cell style and, for sections and subsections, will place a marker at the beginning of the line to draw attention to its status in clarifying the structure of the document. Note also that, as a document is divided into sections and subsections, the bars along the right edge become more numerous, reflecting the divisions created by the inserted sections and subsections.

For example, to add a title to the entire document, we create an insertion bar at the very beginning of the document, select ‘Title’ from the Style submenu of the Format menu, and type
the text of the title. Then, selecting this cell by highlighting its bracket at the right edge of the screen, we can adjust the font, type style, type size, alignment, and several other properties by appropriate selections from submenus accessible in the Format menu.

Mathematica 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 ‘Cut’ from the Edit menu.

- remove all output from the notebook by selecting ‘Delete All Output’ from the Cell 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 (conventional file type .nb) and the directory, and clicking ML on the ‘Save’ button. In a later session, the worksheet can be loaded into Mathematica 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 Mathematica’s notebook interface. You must work with it, explore the menus in the NBI, and read the Mathematica manuals.

8.17 Customizations

When Mathematica is started up, the configuration both of the kernel and of the front-end is defined by a number of default specifications built into the program, by statements in an assortment of system-wide initialization files and by statements in several user-created initialization files. Specifically,\(^{54}\)

- The file $BaseDirectory/Kernel/init.m configures the Mathematica kernel for all users and the file $BaseDirectory/FrontEnd/init.m configures the Mathematica front-end for all users. These files are maintained by the system manager and can be viewed but not edited by individual users.

- The file $UserBaseDirectory/Kernel/init.m modifies the configuration of the kernel established by the system-wide file and must be explicitly created by the individual user.

- The file $UserBaseDirectory/FrontEnd/init.m modifies the configuration of the front-end established by the system-wide file but is not created directly by the user. Instead, Mathematica maintains this file to reflect changes made via statements executed by Mathematica.

These files are simply command files—see Section 8.11—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, for example, variables to which values are assigned and the features defined by packages read in will be available when reading of the file has been completed. As Mathematica is started, the system-wide files are executed and then the user-created file is executed. Thus, the user-created file can be used to override statements in the system-wide file, should that be desired. Together, these files—if they exist at all—define the default configuration set up by Mathematica when finally it is ready to accept input.

\(^{54}\)Here, the system variable $BaseDirectory identifies the top directory in the hierarchy in which the Mathematica structure is stored and the system variable $UserBaseDirectory identifies the top directory in which user-created configuration files are stored. Both of these directories may be hidden.
8.18 Miscellaneous Occasionally Useful Tidbits

In this section, we comment on a number of other features that are occasionally useful. Full details, of course, are available in the *Mathematica* manuals.

### 8.18.1 Turning off Messages

Many of *Mathematica*'s messages can be suppressed by executing the command `Off[...]` and turned back on by executing the command `On[...]`. Here, the ellipses stand for the name of the message. For example, to turn off display of the message that alerts you that a symbol you have just typed is *nearly* the same as some other symbol that you may have intended to type, you can execute the statement

```mathematica
Off[General::Spell1]
```

though full turning off of these messages may also require execution of the statement

```mathematica
Off[General::Spell]
```

These messages can be turned on again with the statements

```mathematica
On[General::Spell1]
On[General::Spell]
```

The system variable `$MessageGroups` stores a list of the existing message groups (the piece of the argument before the double colon).

### 8.18.2 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. Within *Mathematica*, commands for performing these operations should separate directories in a path with the structure that is native to the operating system in use (e.g., forward slash in UNIX, backslash in Windows). For convenience, and to simplify command files to be used with several operating systems, *Mathematica* will translate forward slashes into the appropriate separator for the operating system in use.

### 8.18.3 Default Directory

The default directory or current working directory in *Mathematica* is set when *Mathematica* is launched. Unless an initialization file has been created with a command that changes that directory, it will probably be the user’s home directory or the directory from which *Mathematica* was launched by a command at a *Shell* prompt. The command `Directory[]` will cause the display of the current default directory. As illustrated by the statements

```mathematica
SetDirectory[]
SetDirectory[".."]
SetDirectory["progs"]
SetDirectory["/apps/MATHEM/userprogs"]
```

Set directory to user’s home directory.
Set directory to the parent of current directory.
Set directory to `progs` subdirectory of current directory.
Set directory to indicated directory.
the command *SetDirectory* allows dynamic specification of the current working directory in a variety of ways. Note that the separator between successive directories in a path varies from operating system to operating system. The illustrated separator is the one used in UNIX. Further information about these commands will be found in the *Mathematica* manuals. Further, the statement `FileNames[]` will display a list of the files in the current directory and the statement `FileNames["*.m"]` will display a list of all files in that directory with filetype `.m`. If the file is not in the default directory, then its full path must be specified as part of the filename.

### 8.18.4 Search Path

Whenever an explicit path is not specified for a file that *Mathematica* is to read, the program looks in a sequence of directories defined by the search path. The initial search path is defined in a *Mathematica* initialization file or, in the absence of such a file, by *Mathematica* itself. The initial search path is site-specific and is identified in the *Local Guide*. If not defined by an initialization file, the path is likely to contain, in order, a few specific *Mathematica* directories in the user’s home directory, the default directory, the user’s home directory, and finally a succession of directories descending from the directory `$MATHEMHEAD`. The system variable `$Path` contains a list of the directories in the current search path. Thus, the statement `$Path` will display that list, and statements like

```mathematica
Prepend[ $Path, "/usr/share/CPSUP/mathematica" ]
Append[ $Path, "/usr/local/packages" ]
Insert[ $Path, "/usr/local/temp", 5 ]
```

will create a new value for the list `$Path`, that new value containing all of the elements of the original list in the original order but having the indicated directory at the beginning, at the end, or in the fifth position, respectively. Similarly, statements like

```mathematica
Delete[ $Path, 5 ]
ReplacePart[ $Path, "/usr/local/myfiles", 7 ]
```

will create a new value for the list `$Path` by removing altogether the element in the fifth position or by replacing the element in the seventh position with the indicated directory, respectively.

### 8.18.5 Displaying and Changing Default Options

In the course of introducing several commands, we have learned that the precise effect of many commands depends on the values of associated options. Initially, we accepted *Mathematica*’s defaults. Then we learned that arguments like

```
OptionName -> NewValue
```

(e.g., `PlotRange -> {-1.0,1.0}`) can be used within a command to override the default value for the indicated option, *but only for that execution of the command*. If the default value is to be overridden for *all* subsequent executions of the associated command, however, changing the default with a statement like

```
SetOptions[ CommandName, OptionName -> NewValue ]
```

---

55 See Section 8.18.2.

56 For definiteness, we use the UNIX form for indicating paths. Remember that the precise form in which paths are specified depends on the operating system. The form appropriate at your site is described in the *Local Guide*. 

(e.g., \texttt{SetOptions\[ Plot, \text{PlotRange} \rightarrow \{-1.0,1.0\} \]}) may be more convenient—though this change will, of course, be in effect only until it is changed again (or until a new \textit{Mathematica} session is launched). Finally, the statement

\texttt{Options\[ \text{CommandName} \]}

(e.g., \texttt{Options\[ \text{Plot} \]}) will display a list of all options admitted by the indicated command, each with its current default value.

\subsection*{8.18.6 Placing Text in Graphs}

With suitable commands, we can generate a graphical display to which specified text has been added in the body of the display. Suppose, for example, we wanted to label a graph of the sine function with a string of characters positioned not as a title on the graph as a whole or as a label on an axis but rather as a notation at specified coordinates within the graph. We might use statements like

\begin{verbatim}
In[48]:= Plot[ Sin[x], \{x, 0.0, 10.0\} ];
In[49]:= Graphics[ Text[ "New Graph", \{5.0,0.5\} ] ];
In[50]:= Show[ \{ %48, %49 \} ]
\end{verbatim}

The first of these statements creates a graphics object containing the desired sine curve, the second statement uses the graphics primitive \texttt{Text} to create a second graphics object containing the specified text \textit{centered} at the specified location in the coordinates of the graph and oriented horizontally.\footnote{The \texttt{Graphics} command accepts one or two additional arguments, each of which is a list. For example, the alternative command \texttt{Graphics[ Text[ "New Graph", \{5.0,0.5\}, \{-1.0\}, \{0,1\} ]]} will place the left end of the text at the specified point (third argument) and orient the text vertically and reading from the right side of the graph (fourth argument). Other options for the third and fourth arguments are described in the \textit{Mathematica} manuals.} And the third statement creates and displays a composite graphics object assembled from

- the graphics object created by line \texttt{In[48]} and known by the name \texttt{\%48}, and
- the graphics object created by line \texttt{In[49]} and known by the name \texttt{\%49}.

We could, of course, have nested the first two statements within the third, but the result would have been syntactically difficult to fathom.

Of course, this quick example only hints at the potential of these commands. The \textit{Mathematica} manuals describe many more graphics primitives that can be used as arguments to the \texttt{Graphics} command and also describe numerous options associated both with the \texttt{Graphics} command and with the \texttt{Show} command. Careful study of those items in the \textit{Mathematica} manuals is prerequisite to full exploitation of \textit{Mathematica}’s abilities to create graphics objects and combine them in elaborate displays.

\subsection*{8.18.7 Space Curves}

Sometimes it is desirable to view the trajectory of a particle in three-dimensional space. \textsc{Mathematica} has the ability to plot space curves from knowledge of the trajectory defined parametrically by the functions \(x(t), y(t), z(t)\) giving the coordinates of points on a three-dimensional path. With that information, the command \texttt{ParametricPlot3D} 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

\begin{equation}
\begin{aligned}
x(t) &= 5 \ast \cos t \\
y(t) &= 5 \ast \sin t \\
z(t) &= \alpha t
\end{aligned}
\end{equation}

\footnote{The \texttt{Graphics} command accepts one or two additional arguments, each of which is a list. For example, the alternative command \texttt{Graphics[ Text[ "New Graph", \{5.0,0.5\}, \{-1.0\}, \{0,1\} ]]} will place the left end of the text at the specified point (third argument) and orient the text vertically and reading from the right side of the graph (fourth argument). Other options for the third and fourth arguments are described in the \textit{Mathematica} manuals.}
Figure 8.5: Space curve of a charged particle moving in a constant magnetic field along the \( z \) axis.

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

\begin{verbatim}
In[1]:= alpha = 0.5;

In[2]:= ParametricPlot3D[{5*Cos[t], 5*Sin[t], alpha*t}, {t, 0, 30},
PlotStyle -> {Thickness[0.02], Black}]
\end{verbatim}

will produce the graph in Fig. 8.5.

8.18.8 Directing Plot to Separate Window

The several simple graphics commands we have introduced all direct their output to the window in which the commands are issued. Should you wish to have a new graphics window created for a particular plot, you need only add the Option

\texttt{DisplayFunction \rightarrow CreateDialog}

to whatever else in incorporated in the command. For example, the statement

\begin{verbatim}
Plot[ Tanh[x], \{x, -5.0, 5.0\} ]
\end{verbatim}

produces the specified graph in line while the statement

\begin{verbatim}
Plot[ Tanh[x], \{x, -5.0, 5.0\}, DisplayFunction\rightarrowCreateDialog ]
\end{verbatim}

directs the graph to a newly created (small) separate window. By the second route, the first graph remains on the screen even if the in line graph scrolls off as further statements are executed.
8.19 References

In this chapter, we have introduced only the most important features of Mathematica. A full description of all commands and features is contained in the Mathematica manuals, including

- several documents and resources as described in Section 8.2.

- a number of documents, links to which can be found by entering “Mathematica help” in the search window of your browser. In particular, you should find links to
  
  - the official Mathematica documentation at http://reference.wolfram.com/language/, which brings up on line the same documentation available from the HELP menu in the NBI, and
  
  - many other official and unofficial documents.

- a number of books and on-line materials describing a wide spectrum of uses of Mathematica in a variety of disciplines and for a variety of specific purposes. A list of such books can be found, among other places, from the links http://www.wolfram.com/mathematica/resources/ and http://www.wolfram.com/books/ at the Wolfram Research website or by searching on-line commercial book stores (Amazon, Barnes and Noble, ...) for the keyword Mathematica.

- more specific documents on particular Mathematica commands, links to which can be found by entering “CommandName in MATHEMATICA”, e.g., “integrate in MATHEMATICA” 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

- Getting Started with Mathematica Version # under (operating system), (Wolfram Media, Inc.), which contains installation instructions and a brief description of the issues with which a user of Mathematica must be familiar from the very beginning.

- Stephen Wolfram, The Mathematica Book, Fifth (or earlier) Edition (Wolfram Media, Inc., and Cambridge University Press, 1988, 1991, 1996, 1999, 2003), which contains a tour of the capabilities of Mathematica, an introduction to the practical use of Mathematica, an exhaustive discussion of the design principles underlying Mathematica, a treatment of more advanced topics, several pages of formulae, a section of color plates showing a spectacular array of graphical displays, and a comprehensive set of appendices listing and describing essentially all commands, options, special variables, and other objects that play a role somewhere in Mathematica. This tome of nearly 1500 pages in a single binding is massive and exhaustive.

- Mathematica Version # Standard Add-on Packages (Wolfram Media, Inc., 1996, 1999), which contains detailed descriptions of all components in all of the standard add-on packages.

8.20 Exercises

8.1. Read the Mathematica manuals to find out how the commands Expand, Factor, Apart, and 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
8.20. EXERCISES

8.2. Use Mathematica to convert each of the expressions in the left-hand column in the table below into the expression in the associated right-hand column:

(a) \( \frac{x}{a+b} + \frac{y}{c+d} \) \(\Rightarrow\) \( \frac{(a-b)(c+d) + (a+b)c}{a} \)
(b) \( (x+y)(z+w) \) \(\Rightarrow\) \( a(b+c(d+e)) \)
(c) \( \frac{x}{a+b} + \frac{y}{c+d} \) \(\Rightarrow\) \( \frac{(a-x)^2}{(a^2-2ax+x^2)^{3/2}} \)

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.

8.3. Use Mathematica to convert each of the expressions in the left-hand column in the table below into the expression in the associated right-hand column:

(a) \( \frac{d}{dx} \left( x^2 e^{-x^2} \right) \) \(\Rightarrow\) \(-2x(x^2-1)e^{-x^2}\)
(b) \( \sinh \left( \ln(x + \sqrt{x^2 + a^2}) - \ln(a) \right) \) \(\Rightarrow\) \( \frac{x}{a} \)
(c) \( \frac{1}{x + \sqrt{y}} \) \(\Rightarrow\) \( \frac{x - \sqrt{y}}{x^2 - y} \)

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.

8.4. In Section 8.8.9, 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.
8.5. 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
\[
g(x, t) = \frac{1}{\sqrt{1 - 2xt + t^2}} = \sum_{n=0}^{\infty} P_n(x) t^n
\]
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
\[
\begin{align*}
P_0(x) &= 1 & P_3(x) &= \frac{1}{2}(5x^3 - 3x) \\
P_1(x) &= x & P_4(x) &= \frac{1}{2}(35x^4 - 30x^2 + 3) \\
P_2(x) &= \frac{1}{2}(3x^2 - 1) & P_5(x) &= \frac{1}{2}(63x^5 - 70x^3 + 15x)
\end{align*}
\]
a. Use the generating function and Mathematica’s capabilities for evaluating Taylor series to find the first half-dozen Legendre polynomials, extracting each as an expression bound to a variable. The commands \texttt{Collect} and \texttt{CoefficientList} may be useful.
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. \textit{Hint:} The \texttt{Mathematica} statements might be
\[
\text{In}[29]:= P[0] = 1; P[1] = x;
\]
and then
\[
\text{In}[30]:= PP = (2*n-1)*x*P[n-1]/n - (n-1)*P[n-2]/n
\]
(Verify the expression on the right by using \texttt{Mathematica} 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 \( PP \) depending on \( n \) 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
\[
\text{In}[31]:= P[2] = PP /. n -> 2 \\
\text{In}[32]:= P[3] = PP /. n -> 3 \\
\ldots
\]
c. Find the first half-dozen Legendre polynomials by using the command \texttt{D} to evaluate Rodrigues’ formula. \textit{Hint:} You might find that using a loop would simplify your approach.
d. Be clever and, using either matrices or loops constructed in \texttt{Mathematica}, 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 note that the command \texttt{Integrate} applied to a matrix will automatically apply the command element-by-element.)
e. Within \texttt{Mathematica}, 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. *Mathematica* actually knows quite a bit about many of the important special functions of mathematical physics. In particular, the the \( n\)-th Legendre polynomial as a function of \( x \) is known by the name \texttt{LegendreP[n, x]} and is described fully in the *Mathematica* manuals. Take a look at that documentation and then use that function to determine the first several Legendre polynomials.

8.6. Figure 8.6 shows the circuit diagram for a Wheatstone bridge. Using Kirchhoff’s laws, set up the equations from which you could determine the currents in each branch of the circuit. Then, using *Mathematica*, (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.)

8.7. 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 \( B, D, F, \) and \( C \) 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

\[
\psi(0^-) = \psi(0^+) \quad ; \quad \psi(w^-) = \psi(w^+) \quad ; \quad \left. \frac{d\psi}{dx} \right|_{x=0^-} = \left. \frac{d\psi}{dx} \right|_{x=0^+} \quad ; \quad \left. \frac{d\psi}{dx} \right|_{x=w^-} = \left. \frac{d\psi}{dx} \right|_{x=w^+}
\]

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 *Mathematica’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

\[
R = \left| \frac{B}{A} \right|^2 = \frac{(\kappa^2 + k^2)^2 \sinh^2 \kappa w}{4\kappa^2 k^2 + (\kappa^2 + k^2)^2 \sinh^2 \kappa w} \quad ; \quad T = \left| \frac{C}{A} \right|^2 = \frac{4k^2 \kappa^2}{4\kappa^2 k^2 + (\kappa^2 + k^2)^2 \sinh^2 \kappa w}
\]
Here, the vertical bars symbolize the absolute value of the complex number enclosed by them. (d) Verify that \( R + T = 1 \).

8.8. The distribution of wavelengths \( \lambda \) in the blackbody spectrum at (absolute) temperature \( T \) is given by

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

where \( c \) is the speed of light, \( h \) is Planck’s constant, and \( k \) is Boltzmann’s constant. In terms of the variable \( y = \frac{ch}{\lambda kT} \), this function has the alternative expression

\[
\frac{(ch)^4}{8\pi (kt)^5} u(\lambda, T) = f(y) = \frac{y^5}{e^y - 1}
\]

(a) Verify this transformed form and then, using Mathematica, (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

\[
y - 5 = 0, \quad (y - 5) e^y + 5 = 0
\]

(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.

8.9. Use Mathematica to verify (a) that the solution

\[
x(t) = e^{-\tau} x_0 \left( \cos(at) + \frac{\gamma \sin(at)}{a} \right)
\]

with

\[
a = \sqrt{\omega^2 - \beta^2}; \quad \gamma = \frac{b}{2m}; \quad \omega = \frac{k}{m}
\]

presented in Section 8.8.9 satisfies the desired differential equation and the initial conditions presented at the beginning of that section and (b) that this solution and the solution

\[
y(\tau) = \frac{x_0}{2\sqrt{1 + \beta^2}} \left( e^{-\beta - \sqrt{1 + \beta^2} \tau} + e^{-\beta + \sqrt{1 + \beta^2} \tau} + e^{(-\beta - \sqrt{1 + \beta^2})\tau} \sqrt{1 + \beta^2} 
\]

where \( \tau = \omega t \) and \( \beta = \gamma/\omega \) obtained at the very end of the section agree with each other.

8.10. 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

\[
\int_0^\infty e^{-x} L_n(x) L_m(x) \, dx = 0, \quad \text{all } m < n
\]

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

\[
(n + 1)L_{n+1}(x) = (2n + 1 - x)L_n(x) - n L_{n-1}(x)
\]

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

\[ L_0(x) = 1 \]
\[ L_3(x) = 1 - 3x + \frac{3}{2}x^2 - \frac{1}{6}x^3 \]
\[ L_1(x) = 1 - x \]
\[ L_4(x) = 1 - 4x + 3x^2 - \frac{2}{3}x^3 + \frac{1}{24}x^4 \]
\[ L_2(x) = 1 - 2x + \frac{1}{2}x^2 \]
\[ L_5(x) = 1 - 5x + 5x^2 - \frac{5}{3}x^3 + \frac{5}{24}x^4 - \frac{1}{120}x^5 \]

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 Mathematica statements might be

\[ \text{In}[29]:= L[0] = 1; L[1] = 1-x; \]

and then

\[ \text{In}[30]:= LL = (2*n-1-x)*L[n-1]/n - (n-1)*L[n-2]/n \]
(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 \( \ldots \) with statements like

\[ \text{In}[31]:= L[2] = LL /. n -> 2 \]
\[ \text{In}[32]:= L[3] = LL /. n -> 3 \]
\[ \ldots \]

c. Find the first half-dozen Laguerre polynomials by using the command \( \text{D} \) 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 \( \text{map} \), which will be useful in constructing a single statement that evaluates the integral of each element in a multi-element structure.)

e. Mathematica actually knows quite a bit about many of the important special functions of mathematical physics. In particular, the \( n \)-th Laguerre polynomial as a function of \( x \) is known by the name \( \text{LaguerreL}[n,x] \) and is described fully in the Mathematica manuals. Take a look at that documentation and then use that function to determine the first several Laguerre polynomials.

8.11. Find the eigenvalues and eigenvectors of the matrix

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

8.12. (a) Use the function \( \text{Eigensystem} \) to find the eigenvalues and eigenvectors of the matrix

\[
\begin{pmatrix}
2 & -1 & 0 & 0 \\
-1 & 2 & -1 & 0 \\
0 & -1 & 2 & -1 \\
0 & 0 & -1 & 2
\end{pmatrix}
\]

(b) Use the function \( \text{FullForm} \) to explore the structure of the entity returned by the function \( \text{Eigensystem} \) used in part (a).
8.13. 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 \( n l m | e F | n l' m' \), 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
\[
3e a_0 F \begin{pmatrix}
0 & 0 & -9/2 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -9/2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & -3\sqrt{6} & 0 & -9/\sqrt{3} & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -9/\sqrt{3} & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & -9/2 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & -9/2 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & -9/2 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 
\end{pmatrix}
\]
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 begin by executing the statements
\[
\text{<< LinearAlgebra'\text{MatrixManipulation'}}
\text{mat = ZeroMatrix[4, 4];}
\]
which load the add-on package that defines the command \text{ZeroMatrix} and create a $4 \times 4$ matrix of zeros. Continuing, we then set the nonzero entries with statements like
\[
\text{mat[[1,3]] = desired value}
\]
For verification, the final matrix can be displayed in matrix form with the statement
\[
\text{MatrixForm[ mat ]}
\]

8.14. As shown in Fig. 8.7, 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
\[
KE = \frac{1}{2} m \dot{x}^2 + \frac{1}{2} m \dot{y}^2 \quad ; \quad PE = \frac{1}{2} k \left( \sqrt{(\ell - x)^2 + y^2} - \ell_0 \right) + \ldots
\]
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 Mathematica 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 \[ D[ L, D[x[t],t] ] \] are perfectly acceptable to Mathematica, provided the dependence of \( x \) on \( t \) is explicitly indicated in setting up the Lagrangian \( L \). Thus, for example, in a different problem (particle of mass \( m \) falling under the gravitational attraction of the earth), for which we might set

\[
L = m*D[x[t],t]^2/2 - m*g*x[t]
\]

we can evaluate \( \partial L/\partial \dot{x} \) with the statement

\[
DLDx = D[ L, D[x[t],t] ]
\]

evaluate \( \partial L/\partial x \) with the statement

\[
DLDx = D[ L, x[t] ]
\]

and then, finally, determine the equation of motion with the statement

\[
D[ DLDx, t] \) \] - DLDx == 0
\]

8.15. In the Cartesian coordinate system illustrated in Fig. 8.8, 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

$$KE = \frac{1}{2}m_1\dot{x}_1^2 + \frac{1}{2}m_1\dot{y}_1^2 + \frac{1}{2}m_2\dot{x}_2^2 + \frac{1}{2}m_2\dot{y}_2^2; \quad PE = m_1gy_1 + m_2gy_2$$

use Mathematica 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.

8.16. 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 Mathematica to evaluate these integrals to $O(k^6)$ by expanding the integrands in Taylor series before evaluating the integrals.

8.17. 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) $\hat{f}(s) = \frac{a + bs}{s^2 + \omega^2}$  (f) $\hat{f}(s) = \frac{a}{(s^2 + 9\omega^2)(s^2 + 4\omega^2)(s^2 + \omega^2)}$

In (c), you may have to express the hyperbolic cosine in exponential form while in (f) you may have to help Mathematica’s routine InverseLaplaceTransform by first invoking the command Apart to expand the desired function in partial fractions.

8.18. 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 $\theta = 39^\circ, 40^\circ,$ and $41^\circ$ are 0.7251744, 0.729383, and 0.7258887, respectively.

8.19. 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

$$\int_{x_1}^{x_3} y(x) \, dx = \frac{\delta x}{3} (y_1 + 4y_2 + y_3)$$

8.20. 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.
8.21. Using *Mathematica*, 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

\[ \frac{U(x)}{U_0} = -\frac{1 + \bar{x}^2}{8 + \bar{x}^4} \]

You should be able to find a similar dimensionless version of the expression giving the force.

8.22. Using *Mathematica*, find complete (symbolic) solutions to each of the following problems, and use *Mathematica* also to verify that the solutions you obtain actually do satisfy the original ODE and initial conditions.

a. \[ \frac{d^2 x}{dt^2} = a, \quad x(0) = x_0, \quad v(0) = v_0, \]

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. \[ m \frac{d^2 x}{dt^2} = -\varepsilon E_0 \cos(\omega t + \theta), \quad x(0) = x_0, \quad v(0) = v_0, \]

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. \[ m \frac{d^2 x}{dt^2} = -mg + b \left( \frac{dx}{dt} \right)^2, \quad x(0) = 0, \quad v(0) = 0, \]

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

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

for the motion of a projectile moving under gravity in a viscous medium when the motion starts at the origin with the initial velocity \( \mathbf{v} = v_x 0 \hat{i} + v_z 0 \hat{k} \).

8.23. Using the functions *Grad*, *Laplacian*, *Div*, and *Curl* defined in Section 8.8.10,

a. Evaluate the gradient of (i.e., the negative of the force field associated with) each of the functions

\[ V_1(x, y, z) = \frac{1}{(x^2 + y^2 + z^2)^{1/2}} \quad V_2(x, y, z) = \frac{z}{(x^2 + y^2 + z^2)^{3/2}} \]

\[ V_3(x, y, z) = \frac{2z^2 - x^2 - y^2}{(x^2 + y^2 + z^2)^{3/2}} \quad V_4(x, y, z) = \frac{e^{-a(x^2 + y^2 + z^2)^{1/2}}}{(x^2 + y^2 + z^2)^{3/2}} \]

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 \( \mathbf{r} = x \hat{i} + y \hat{j} + z \hat{k} \), show (1) that \( \nabla \times \mathbf{r} = 0 \) and (2) that \( \nabla \cdot \mathbf{r} = 3 \).

d. With \( V_5(x, y, z) = x^2 + y^2 + z^2 \), show that \( \nabla^2 V_5(x, y, z) = \nabla \cdot \nabla V_5(x, y, z) = 6 \).

e. Show that \( \nabla^2 V_1(x, y, z) = 0 \). (Note: Except at \( x = y = z = 0 \).)

f. Evaluate the Laplacian of \( V_4(x, y, z) \), \( \nabla^2 V_4(x, y, z) \).

8.24. Using the functions *Grad*, *Laplacian*, *Div*, and *Curl* defined in Section 8.8.10,
a. Evaluate the gradient of (i.e., the negative of the force field associated with) each of the functions
\[ V_1(r, \phi, z) = \frac{1}{(r^2 + z^2)^{1/2}} \]
\[ V_2(r, \phi, z) = \frac{z}{(r^2 + z^2)^{3/2}} \]
\[ V_3(r, \phi, z) = \frac{2z^2 - r^2}{(r^2 + z^2)^{3/2}} \]
\[ V_4(r, \phi, z) = \frac{e^{-a(r^2 + z^2)^{1/2}}}{(r^2 + z^2)^{1/2}} \]
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 \( V_4(r, \phi, z) \), \( \nabla^2 V_4(r, \phi, z) \).

8.25. Using the functions Grad, Laplacian, Div, and Curl defined in Section 8.8.10,

a. Evaluate the gradient of (i.e., the negative of the force field associated with) each of the functions
\[ V_1(r, \theta, \phi) = \frac{1}{r} \]
\[ V_2(r, \theta, \phi) = \frac{z}{r^2} \]
\[ V_3(r, \theta, \phi) = \frac{r^2(3 \cos^2 \theta - 1)}{r^3} \]
\[ V_4(r, \theta, \phi) = \frac{e^{-a/r^{1/2}}}{r} \]
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 \( \nabla \cdot r = 3 \).

8.26. Create four three-component vectors \( A, B, \) and \( C \) with statements like
\[ A = \{A1, A2, A3\}; B = \{B1,B2,B3\}; CC = \{CC1,CC2,CC3\}; DD = \{DD1,DD2,DD3\}; \]

Then, using the file crossdot.m as described in Section 8.11, show that

a. \( A \times B = -B \times A \), or, equivalently, that \( A \times B + B \times A = 0 \).

b. \( A \times (B \times CC) = (A \cdot CC)B - (A \cdot B)CC \)

c. \( (A \times B) \times CC = (A \cdot CC)B - (B \cdot CC)A \)

d. \( A \cdot (B \times CC) = (A \times B) \cdot CC \)

e. \( (A \times B) \cdot (CC \times DD) = (A \cdot CC)(B \cdot DD) - (A \cdot DD)(B \cdot CC) \)

---

58 The variables \( C \) and \( D \) in MATHEMATICA are protected.
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.
CHAPTER 9. INTRODUCTION TO PROGRAMMING

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 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.

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.

2Some languages are case-sensitive and others are not. For our general discussion, we shall assume case insensitivity, so A and a are indistinguishable in variable names.
9.1. COMPONENTS OF A PROGRAMMING LANGUAGE

Even in the generic discussion of this section, however, we shall pay attention to the data type of each quantity represented. In Chapter 1, we distinguished floating point numbers, integers, and character strings from one another in several ways. In structuring programs, we must remain always aware of this distinction. Further, the computer in interpreting or compiling our source code must either make assumptions or be told the data type of each variable we use. Some languages (e.g., Pascal, C) require an explicit declaration of variable names and associated data types before the name can be used. Other languages (e.g., BASIC, FORTRAN) exploit implicit data typing by looking to the composition of the variable name to determine the intended data type—though more recent versions of FORTRAN, for example, also admit explicit data typing. Still others (e.g., IDL, MATLAB, OCTAVE) assign a data type dynamically and automatically on the basis of the value assigned to the variable. In our general discussion, we shall adopt a convention that keeps us consciously aware of data type every time we use a variable name. Following the pattern actually used in BASIC, we shall take variable names with no appended suffix to represent floating point numbers, variable names with an appended percent sign % to represent integers, and variable names with an appended dollar sign $ to represent character strings. Occasionally, we shall use an appended at sign @ to stand for any one of these three possibilities (no suffix, %, or $). Thus, in the pseudocode we are defining, the names x, position, and field identify memory cells for floating point values, the names I%, count%, and lower_limit% are valid names for cells storing integers, the names word$ 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 × 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 × n matrix), or even higher dimensional. As a structured entity, an array is identified by a single name, e.g., data, values%, or names$, depending on the data type of the elements of the array. Individual elements are identified by attaching an integer index or indices to the array name, e.g., data(I%, J%), values%(4%), or names$(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

```
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:

\[
\langle \text{variable} \rangle ←\langle \text{expression determining value to be assigned} \rangle
\]

e.g.,

\[
\begin{align*}
\text{val} & ← \sin(\pi x) \\
\text{count}\% & ← \text{count}\% + 1\% \\
\text{first}\_\text{name}$ & ← "David"
\end{align*}
\]

e.t.c. Any valid arithmetic expression utilizing the symbols + for addition (or string concatenation), - for subtraction, * for multiplication, / for division, and ^ for exponentiation can appear on the right hand side of the assignment symbol ←. 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:

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

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

### 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, …). The six \textsl{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.

\footnote{We declare that integer \textit{constants} shall be explicitly identified with a trailing percent sign and that string \textit{constants} shall be enclosed in double quotation marks. Some languages use single quotation marks, while others will accept either (provided they are paired).}
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>true 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 \( \langle \text{condition} \rangle \) is false and false if \( \langle \text{condition} \rangle \) 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

\(^5\)We 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

\[
\text{val ← sin(pi*x); count% ← count% + 1%; first_name$ ← "David"}
\]

• 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

\[
\text{U(I%, J%) ← 0.25 * ( U(I%+1%, J%) + U(I%-1%, J%) → } \\
\text{+ U(I%, J%+1%) + U(I%, J%-1%) )}
\]

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

- Sequence, which is conveyed by the order of the statements in the algorithm.
- Two-way selection, expressed generically with a statement like\(^8,9\)

\[\text{CNT% ← 0% ! Set CNT% to 0%}\]

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

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

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

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

\[\text{CNT% ← 0% ! Set CNT% to 0%}\]

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⟩ D0 ⟨block1 of statements⟩
    OF ⟨condition2⟩ D0 ⟨block2 of statements⟩
    OF ⟨condition3⟩ D0 ⟨block3 of statements⟩
    :
    OF OTHERS D0 ⟨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

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

11 Not all compilers adopt this strategy. Some will execute the blocks associated with every true condition in the structure.
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.,

\[ \text{WHILE (condition) DO (block of statements)} \]

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

\[ \text{REPEAT (block of statements) UNTIL (condition)} \]

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

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

\[ \text{FOR I\% \leftarrow IMIN\% THRU IMAX\% DO (block of statements)} \]

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

\[ \text{FOR I\% \leftarrow IMIN\% THRU IMAX\% STEP INC\% DO (block of statements)} \]

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 IMIN\% equals IMAX\% and not at all if IMIN\% exceeds IMAX\%—but beware; different languages may behave differently in this regard.

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

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

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

1. Exchange the values stored in \( A @ \) and \( B @ \):\(^{12}\)

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

The temporary memory cell \( \text{TEMP} @ \) 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]:

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

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

3. Stopping a loop by counting up:

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

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:

\[
\begin{align*}
\text{TIMES} _{\text{REMAINING}} @ & \leftarrow \langle \text{desired number of executions} \rangle \\
\text{LOOP} \\
& \text{EXIT LOOP WHEN TIMES} _{\text{REMAINING}} @ = 0 \% \\
& \langle \text{block of statements} \rangle \\
& \text{TIMES} _{\text{REMAINING}} @ \leftarrow \text{TIMES} _{\text{REMAINING}} @ - 1 \%
\end{align*}
\]

\(^{12}\)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

\[
\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} = 0
\]

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, \ldots. Beyond the partial differential equation itself, a complete problem requires the statement of boundary
conditions—often the stipulation of the value of \( U \)—at all points on the boundary of the region in which a solution is sought. Thus, for example, a complete problem might seek a solution to the Laplace equation in a square, subject to the requirement that the solution assume the value 0.0 on three edges of the square and the value 100.0 on the fourth edge, as shown in Fig. 9.3(a). Physically, this solution would convey the temperature within the square when three of its edges are maintained at 0° and the fourth edge is maintained at 100° or the electrostatic potential within the square when three edges are maintained at a potential of 0 Volts and the fourth edge is maintained at a potential of 100 Volts.

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

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

(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—involves guessing a starting solution and then refining that solution iteratively by stepping systematically and repeatedly through the grid of interior points, replacing the value at each point by the average of the values at its four nearest neighbors. Each pass through the entire grid constitutes one iteration, and we keep going—iteration after iteration—until we are satisfied.
that the process has converged satisfactorily.\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

\begin{verbatim}
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
\end{verbatim}

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

\begin{verbatim}
DIMENSION U(15%,15%)
FOR I% ← 1% THRU 15% DO
  FOR J% ← 1% THRU 15% DO U(I%, J%) ← 0.0
FOR J% ← 1% THRU 15% DO U(15%, J%) ← 100.0
FOR ITCNT% ← 1% THRU 30% DO Conduct one iteration
  FOR I% ← 2% THRU 14% DO
    FOR J% ← 2% THRU 14% DO
      U(I%, J%) ← 0.25 * ( U(I%+1%, J%) → + U(I%-1%, J%) + U(I%, J%+1%) → + U(I%, J%-1%) )
  FOR I% ← 1% THRU 15% DO
    FOR J% ← 1% THRU 15% DO WRITE U(I%, J%)
\end{verbatim}

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></td>
</tr>
<tr>
<td>07:</td>
<td>2</td>
<td>(n) (number of arrays)</td>
</tr>
<tr>
<td>08:</td>
<td>3</td>
<td>(n_x)</td>
</tr>
<tr>
<td>09:</td>
<td>1</td>
<td>(n_y)</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
- **\(n_x n_y n_z\)** lines specifying the values in the first array—one value per line in the order that results from the generic coding

\[
\text{FOR K\% ← 1\% THRU NZ\% DO} \\
\text{FOR J\% ← 1\% THRU NY\% DO} \\
\text{FOR I\% ← 1\% THRU NX\% DO WRITE A(I\%,J\%,K\%)}
\]

—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.

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

\[
\text{DIMENSION ARRAY}((n_x,n_y))
\]

\[
\text{! Prepare array for values}
\]

\[
\text{NARR\%} \leftarrow 1\%
\]

\[
\text{! Set number of arrays}
\]

\[
\text{NX\%} \leftarrow n_x
\]

\[
\text{! Set number of grid points}
\]

\[
\text{NY\%} \leftarrow n_y
\]

\[
\text{! in each coordinate}
\]

\[
\text{NZ\%} \leftarrow 1\%
\]

\[
\text{! Set increments}
\]

\[
\text{DX} \leftarrow (x_{\text{max}} - x_{\text{min}})/(n_x - 1)
\]

\[
\text{DY} \leftarrow (y_{\text{max}} - y_{\text{min}})/(n_y - 1)
\]

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

\[
\text{FOR J\%} \leftarrow 1\% \text{ THRU NY\% DO}
\]

\[
\text{BEGIN_BLOCK}
\]

\[
\text{YF} \leftarrow y_{\text{min}} + (J\% - 1\%) \times DY
\]

\[
\text{FOR I\%} \leftarrow 1\% \text{ THRU NX\% DO}
\]

\[
\text{BEGIN_BLOCK}
\]

\[
\text{XF} \leftarrow x_{\text{min}} + (I\% - 1\%) \times DX
\]

\[
\text{ARRAY(I\%,J\%)} \leftarrow f(XF,YF)
\]

\[
\text{END_BLOCK}
\]

\[
\text{END_BLOCK}
\]

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

\[
\text{ATTACH FILE (filename) ON CHANNEL 1 FOR WRITING}
\]

\[
\text{WRITE TO CHANNEL 1, "Line of explanation (Title, Author, Date?)"}
\]

\[
\text{WRITE TO CHANNEL 1, "Line of explanation"}
\]

\[
\text{WRITE TO CHANNEL 1, "***"}
\]

\[
\text{WRITE TO CHANNEL 1, "***"}
\]

\[
\text{WRITE TO CHANNEL 1, NARR\%}
\]

\[
\text{WRITE TO CHANNEL 1, NX\%}
\]

\[
\text{WRITE TO CHANNEL 1, NY\%}
\]

\[
\text{WRITE TO CHANNEL 1, NZ\%}
\]

\[
\text{FOR J\%} \leftarrow 1\% \text{ TO NY\% DO FOR I\%} \leftarrow 1\% \text{ TO NX\% DO WRITE TO CHANNEL 1, ARRAY(I\%,J\%)}
\]

\[
\text{CLOSE FILE ON CHANNEL 1}
\]

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

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

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

\begin{verbatim}
ATTACH FILE ⟨filename⟩ ON CHANNEL 1 FOR READING
READ FROM CHANNEL 1, Comma-separated list of variables for storage of values
  :
CLOSE FILE ON CHANNEL 1
\end{verbatim}

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

\textit{Note:} All FORTRAN programs (*.f) and FORTRAN-created data files (*.f.dat)
referred to in this chapter are available in the directory $\texttt{HEAD/fortran}$, where (as
defined in the Local Guide) $\texttt{HEAD}$ 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:

\begin{itemize}
  \item Except for characters within quoted strings, FORTRAN is insensitive to case.
  \item 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.
  \item Legal variable names in FORTRAN begin with a letter, which may be followed by any number
  of characters chosen from the letters $\text{A-Z}$, the numeric digits $0-9$, and—in extended FORTRAN-
  77—the underscore, the dollar sign $\$\text{,}$ and the period ‘.’. Note, however, that in \textit{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.
\end{itemize}
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

```
REAL IMAP, LETTER(20)
DOUBLE PRECISION X, Y
CHARACTER A
INTEGER ROOT, Z(30,50)
```

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

```
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 `PROGRAM ⟨ProgramName⟩` and concludes with the statement `END`.

• The statement beginning `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 not variables, since they cannot be assigned different values during execution of the program. The `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 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 `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 `.gt.`, `.ge.`, `.lt.`, `.le.`, `.eq.`, and `.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 `.and.`, `.or.`, and `.not.` symbolize AND, OR, and NOT, respectively.

• A FORTRAN loop is introduced with the keyword `DO` and terminated with the keyword `ENDDO`. The loops in this sample program are counted loops, a feature conveyed by following the keyword `DO` with an 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).\(^\text{19}\) Then, in the line to be output, there will be fifteen—the 15 in the designation \texttt{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.\(^\text{20}\) 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.\(^\text{21,22}\)

• 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

\[
\begin{align*}
\text{DO } J &= 1, \text{ NYDIM} \\
\text{DO } I &= 1, \text{ NXdIM} \\
\text{WRITE}( \text{ UNIT = 6, FMT = 200 } ) &\ U(I,J) \\
\text{ENDDO} \\
\text{ENDDO}
\end{align*}
\]

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 (\texttt{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., \texttt{4I10} 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., \texttt{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 \texttt{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.
9.5. ADDITIONAL FEATURES OF FORTRAN

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 )
```

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.

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.

---

23 Throughout 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.

24 Incidentally, note that, if necessary, execution of a FORTRAN program can usually be aborted by typing ⟨CONTROL/C⟩.
IF \langle condition1 \rangle THEN
\langle block1 of statements \rangle
ELSE IF \langle condition2 \rangle THEN
\langle block2 of statements \rangle
: 
ELSE IF \langle conditionn \rangle THEN
\langle blockn of statements \rangle
ELSE
\langle catch-all block of statements \rangle
END IF

Test \langle condition1 \rangle.
Execute \langle block1 \rangle and exit when \langle condition1 \rangle is true.
Test \langle condition2 \rangle (if \langle condition1 \rangle is false).
Execute \langle block2 \rangle and exit when \langle condition2 \rangle is true.
Test \langle conditionn \rangle (if \langle condition(n-1) \rangle is false).
Execute \langle blockn \rangle and exit when \langle conditionn \rangle is true.
Execute \langle catch-all block \rangle and exit when all conditions are false.

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

\[
\text{IF} \quad \langle \text{condition} \rangle \quad \langle \text{Single FORTRAN statement} \rangle
\]

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 \textit{write} to print a prompting message conveying the item wanted and then the command \textit{read} to read that item from the keyboard and store it in a suitable variable with statements like

\[
\text{WRITE( *, * ) 'Suitable prompting message: '}
\text{READ( *, * ) VALUE}
\]

where we have allowed the compiler to choose the default input and output devices (first asterisk in each statement; unit 6 for the \textit{write} statement; unit 5 for the \textit{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

\[
\text{WRITE( UNIT=*, FMT=* ) 'Suitable prompting message: '}
\text{READ( UNIT=*, FMT=* ) VALUE}
\]

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:

\[25\text{Beyond the formats } F, E, \text{ and } I \text{ described in footnote 22, the descriptor } A \text{ conveys to the compiler that it is to read characters from the input line. If only } A \text{ is specified, the compiler reads and counts characters until } \langle \text{RETURN} \rangle \text{ is typed; if the format descriptor takes the form } A^n \text{, the compiler reads } n \text{ characters.}\]
9.5. ADDITIONAL FEATURES OF FORTRAN

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

```fortran
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

26 Check your local FORTRAN manual for the features of the FORTRAN compiler available at your site.
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

```fortran
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

```fortran
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

```fortran
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.

### 9.6 Solving Laplace’s Equation with C

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

Rather than systematically laying out features of C as a programming language before presenting a program, we elect to present a simple program illustrating several of those features and then comment thereon. A C program implementing the algorithm laid out in Section 9.3.1 for solving Laplace’s equation is presented in Table 9.5. Note the following about this program:

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

Table 9.5: A C program to solve Laplace’s equation.

```c
/* 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++)
      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( "%.2f", U[i][j] );
      printf( "\n" );
    }
}
```
• 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.

• 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,
9.6. SOLVING LAPLACE’S EQUATION WITH C

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

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

---

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, requiring in both 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.

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.
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.\footnote{Incidentally, note that, if necessary, execution of a C program can usually be aborted by typing (CONTROL/C).}

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

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

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

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

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

### 9.7 Additional Features of C

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

#### 9.7.1 Multi-Way Selection

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

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

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

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

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

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

9.7.2 Reading Data from Keyboard

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

printf( "\nSuitable prompting message: " );
scanf( "%f", &value );
(Note the ampersand preceding the variable name; it makes sure that the argument refers explicitly to the address of the memory cell(s) in which the value of the variable is to be stored and must be present for proper functioning.) Finally, note that, when coded like

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

9.7.3 Additional Loops

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

```c
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

```c
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
9.11 SOLVING LAPLACE’S EQUATION WITH PYTHON

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

\texttt{return\ (expression\ giving\ value\ to\ be\ returned)}

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

\texttt{VarName = FuncName( Arg1, Arg2, ... )}

where \texttt{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 \texttt{without} using the “official” arguments. To do so in C, we must make use of \texttt{global variables}, which are declared by placing the declarations early in the listing and, in particular, \texttt{outside} and \texttt{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.

9.11 Solving Laplace’s Equation with PYTHON

\textit{Note}: All PYTHON programs (\texttt{*.py}) and FORTRAN-created data files (\texttt{*.py.dat}) referred to in this chapter are available in the directory $\texttt{HEAD/python}$, where (as defined in the \textit{Local Guide}) $\texttt{HEAD}$ must be replaced by the appropriate path for your site.

We have already laid out many features of PYTHON as a programming language in Chapter 5. We elect here to present a simple program and then comment on any needed features not already introduced. Implementing the algorithm laid out in Section 9.3.1, a simple program for solving Laplace’s equation and storing the results in a file is presented in Table 9.6. Note the following about this program:

- The \texttt{for} loop was introduced in Section 5.4. Remember, in particular that the first line in a loop ends with a \texttt{colon} and that indentation of the body of a loop is critical to conveying the bounds of the loop to the PYTHON interpreter.

- The formatting of the output illustrated in the innermost statement in the segment ‘Display solution on the screen’ was introduced in Section 5.6.3. The comma terminating the first \texttt{print} statement in this innermost statement suppresses the automatic line feed at the end of each execution of that statement. The simpler statement \texttt{print(U)} would also display the solution on the screen, though the resulting output (especially if we had used a much large array for \texttt{U}) might be quite uninterpretable.
Table 9.6: A PYTHON program to solve Laplace’s equation.

```python
# laplace.py

# 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.

import numpy as np  # Import needed packages
xdim = 15; ydim = 15; maxit = 300  # Set parameters

U = np.zeros((xdim, ydim))

# ***** Initialize U(i,j); set boundary conditions *****
for j in np.arange(ydim): U[xdim-1,j] = 100.0

# ***** Iterate to solution *****
for itcnt in np.arange(maxit):
    for i in np.arange(1, xdim-1):  # Conduct one iteration
        for j in np.arange(1, ydim-1):
            U[i,j] = 0.25 * ( U[i+1,j] + U[i-1,j] + U[i,j+1] + U[i,j-1] )

# ***** Display solution on screen *****
for j in np.arange(ydim):
    for i in np.arange(xdim):
        print('{0:7.2f}'.format(U[i,j])),
    print

# ***** Write solution to a file *****

f = open('laplace_py.dat', 'w')
for j in np.arange(ydim):
    for i in np.arange(xdim):
        f.write('{0:7.2f}'.format(U[i,j]))
        f.write('
')
f.close()
```

- Anticipating that we might want subsequently to import this solution to a different tool for examination, the last segment in this program writes the solution to a file in the default directory.

- Whether we write the file or not, we could also have displayed the solution as a mesh diagram on the screen by adding after `U` has been computed the statements\(^{35}\)
  \[xx = np.linspace(0.0, 1.0, 15); yy = np.linspace(0.0, 1.0, 15)\]
  \[x, y = np.meshgrid(xx, yy)\]

\(^{35}\)These commands were first described in Section 5.11.
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure()
ax = plt.axes(projection='3d')
ax.plot_wireframe(x, y, U, color='black')
plt.show()

This program is stored in a file named laplace.py in the directory $HEAD/python. Once that file has been copied to the default directory, it can be run with a command as described in the Local Guide. In particular, in some installations, the command

python laplace.py

issued at an MS-DOS or UNIX/LINUX command window, the statement

execfile('laplace.py') or exec(open('laplace.py').read() )

issued at the PYTHON prompt >>>, or selecting ‘Run Module’ from the RUN MENU or pressing F5 in the PYTHON Edit Window will execute the program and create the file laplace.py.dat in the default directory.34

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.35

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 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 lines36

\[ \text{Incidentally, remember that, if necessary, execution of a PYTHON program can usually be aborted by typing (CONTROL/C).} \]

35Note 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.

36In 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.
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

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

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 9.7. A full listing of this program will be found in Section 9.A. The program itself is stored in the file 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 Local Guide. In UNIX, those commands might be

f77 -o irrad.xf irrad.f
./irrad.xf

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

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 9.8. Then we set appropriate values for a few constants with the lines

Program notes: (1) With IX=50, the loop controlled by the statement DO I = 1, IX is executed first for I=1, then for I=2, 1=3, ..., continuing until it has been executed for I=50, which is the index of the IXth element. (2) With I and J ranging from 1 to 50 and DX and DY defined as 6π/49, XF and YF range from −3π to 3π 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., R**T evaluates r^t.

In 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.
Table 9.7: 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.

```
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
```

\[
\text{threepi} = 3.0 * 3.1415926535; \quad /* \text{Set constant} */
\]
\[
dx = 2.0 * \text{threepi} / (\text{xdim} - 1.0); \quad /* \text{Set increments} */
\]
\[
dy = dx;
\]

Next, we evaluate the irradiance given by Eq. (9.4) with the lines\(^{39}\)

---

\(^{39}\)Program notes: (1) With xdim=50, the loop controlled by the statement \(\text{for}( x=0; x<\text{xdim}; x++ )\) is executed first for \(x=0\), then for \(x=1\), \(x=2\), ..., continuing until it has been executed for \(x=49\), which is the index of the xdim\(_{th}\) element. (2) With \(x\) and \(y\) ranging from 0 to 49 and \(dx\) and \(dy\) defined as \(6\pi/49\), \(xf\) and \(yf\) range from \(-3\pi\) to \(3\pi\) in 49 equal steps. (3) Note the position of the statement evaluating \(yf\). For computational efficiency, it has been placed in the outer loop. (4) The function \(\text{pow}(r,t)\) evaluates \(r^t\).
Table 9.8: Beginning lines of program irrad.c.

/* PROGRAM irrad.c */

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

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

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

Finally, we open the file, write the (nine) heading lines and then the irradiance into the file, and close the file with the lines listed in Table 9.9 A full listing of this program will be found in Section 9.B. The program itself is stored in the file irrad.c in the directory $HEAD/cc. Once that file has been copied to the default directory, it can be compiled, linked, and run with commands as described in the Local Guide. In UNIX, those commands might be

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

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

9.12.6 ... with PYTHON

A PYTHON program 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, needed modules are imported, 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
9.12. CREATING AND STORING TWO-DIMENSIONAL SCALAR ARRAYS

Table 9.9: 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<ylim; y++ )
{
    for( x=0; x<xlim; x++ )
    {
        fprintf( fptr,"%e\n", I[x][y] );
    }
}

fclose( fptr );
```

program, (2) specify parameters at the beginning 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.

```python
# irrad.py
import numpy as np
narr = 1; ix = 50; iy = 50; iz = 1  # Set dimensions
I = np.zeros( (ix,iy) )  # For irradiance
threepi = 3.0*np.pi  # Set constant
dx = 2.0*threepi/(ix-1.0)  # Set increments
dy = dx
```

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

---

40 In 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.

41 Program notes: (1) With \texttt{ix=50}, the loop controlled by the statement \texttt{for \texttt{i} in np.arange(\texttt{ix})} is executed first for \texttt{i=0}, then for \texttt{i=1}, \texttt{i=2}, ..., continuing until it has been executed for \texttt{i=49}, which is the index of the \textit{i}th element. (2) With \texttt{i} and \texttt{j} ranging from 0 to 49 and \texttt{dx} and \texttt{dy} defined as \(6\pi/49\), \texttt{xf} and \texttt{yf} range from \(-3\pi\) to \(3\pi\) in 49 equal steps. (3) Note the position of the statement evaluating \texttt{yf}. For computational efficiency, it has been placed in the \textit{outer} loop.
Table 9.10: Closing lines of program irrad.py.

```python
f = open( 'irrad_python.dat', 'w' )
f.write( 'Irradiance; Author: David M. Cook; Date: 30 August 2018\n' )
f.write( 'Fraunhofer diffraction at square aperture\n' )
f.write( 'Program described in CPSUP\n' )
f.write( '***\n' )
f.write( '***\n' )
f.write( str(narr)+'\n' )
f.write( str(ix)+'\n' )
f.write( str(iy)+'\n' )
f.write( str(iz)+'\n' )
for j in np.arange(iy):
    for i in np.arange(ix):
        f.write( str( I[i,j] ) + '\n' )
f.close()
```

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 9.10. A full listing of this program will be found in Appendix 9.C. The program itself is stored in the file `irrad.py` in the directory `$HEAD/python`. Once that file has been copied to the default directory, it can be executed with commands as described in the Local Guide. In particular, in some installations, executing the statement

```python
python irrad.py
```

at an MS-DOS or UNIX/LINUX command window, the executing the statement

```python
execfile('irrad.py') or exec(open('irrad.py').read() )
```

at the PYTHON prompt `>>>`, or selecting ‘Run Module’ or pressing F5 in the PYTHON Edit Window will run the program and create the file `irrad_python.dat` in the default directory.

### 9.13 Creating and Storing Three-Dimensional Scalar Arrays

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

$$ p(x, y, z) = \frac{1}{2\pi(27)^{3/2}} \rho^{3 e^{-2\rho/3}} \left( \frac{3z^2}{\rho^2} - 1 \right)^2 = \frac{1}{2\pi(27)^{3/2}} e^{-2\rho/3} (9z^4 - 6z^2\rho^2 + \rho^4) \quad (9.5) $$
where the coordinates are all measured in units of the Bohr radius and

\[ \rho = \sqrt{x^2 + y^2 + z^2} \]  (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.D, is named `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 \([X_0, Y_0, Z_0]\) to its final value \([X_0, Y_0, Z_0] + [X_{RANGE}, Y_{RANGE}, Z_{RANGE}]\). 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, `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.\(^{42}\) Once the program has been executed, the file `pdens.f.dat` will exist in the default directory.

### 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.E, is named `pdens.c` and contains sections

\(^{42}\)For instructions on how to compile and run the program, see Section 9.12.1 and the Local Guide.
• 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 \( x_f, y_f, \) and \( z_f \), we introduce symbols \([x_{\text{range}}, y_{\text{range}}, z_{\text{range}}]\) for the interval on each axis and \([x_0, y_0, z_0]\) 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 \( x_s, y_s, \) and \( z_s \) for the squares of the coordinates and \( \rho \) and \( \rho s \) 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 \([x_0, y_0, z_0]\) to its final value \([x_0, y_0, z_0] + [x_{\text{range}}, y_{\text{range}}, z_{\text{range}}]\). 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, \( \text{pdens.c} \) 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.\(^ {43} \) Once the program has been executed, the file \( \text{pdens_c.dat} \) will exist in the default directory.

\(^{43}\)For instructions on how to compile and run the program, see Section 9.12.2 and the Local Guide.

\(^{44}\)For instructions on how to compile and run the program, see Section 9.12.6 and the Local Guide.

9.13.6 . . . with PYTHON

A PYTHON program for creating and storing the necessary three-dimensional scalar array is similar to the program presented in Section 9.12.6. 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 Appendix 9.F, is named \( \text{pdens.py} \) and contains sections

• Identifying the program, importing necessary modules, and setting values for several constants. 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 \( x_s, y_s, \) and \( z_s \) for the squares of the coordinates and \( \rho \) and \( \rho s \) 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 \([x,y,z]_0\) to its final value \([x,y,z]_0 + [x,y,z]\)range. 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, \( \text{pdens.py} \) can be copied from the directory \$\text{HEAD/python} \) into your own directory, executed or—should this be necessary for your application—edited before being executed to create the desired file.\(^ {44} \) Once the program has been executed, the file \( \text{pdens_python.dat} \) will exist in the default directory.

\(^{43}\)For instructions on how to compile and run the program, see Section 9.12.2 and the Local Guide.

\(^{44}\)For instructions on how to compile and run the program, see Section 9.12.6 and the Local Guide.
9.14 Creating and Storing Two-Dimensional Vector Arrays

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

As an example of a two-dimensional vector field, we choose the magnetic field in a transverse electric (TE) electromagnetic wave propagating in a waveguide with perfectly conducting walls (though we choose also to compute and save the associated electric field). As shown in Fig. 9.4, we take the waveguide to be oriented so that the wave propagates in the positive \( x \) direction. Let the guide have a rectangular cross-section with dimensions \((b, d)\), i.e., \((0 < y < b, 0 < z < d)\). Further, let the wave be polarized in the \( z \) direction and consider the particular TE waves whose electric field does not depend on \( z \). In mksa units, the (complex) fields in this guide are given by\(^{45}\)

\[
\mathbf{E}(\mathbf{r}, t) = A \sin \frac{n \pi y}{b} e^{i(\kappa_x x - \omega t)} \mathbf{k} \tag{9.7}
\]
\[
\mathbf{H}(\mathbf{r}, t) = \nabla \times \mathbf{E} = \frac{A}{i \omega \mu_0} \left( \frac{n \pi}{b} \cos \frac{n \pi y}{b} \mathbf{i} - i \kappa_x \sin \frac{n \pi y}{b} \mathbf{j} \right) e^{i(\kappa_x x - \omega t)} \tag{9.8}
\]

where \( n = 1, 2, 3, \ldots \) and, with \( c \) standing for the speed of light,

\[
\kappa_x^2 = \frac{\omega^2}{c^2} - \left( \frac{n \pi}{b} \right)^2 \tag{9.9}
\]

If we focus on the physical fields (real parts of \( \mathbf{E} \) and \( \mathbf{H} \)) specifically at \( t = 0 \), we find that

\[
\mathbf{E}(\mathbf{r}) = A \cos \kappa_x x \sin \frac{n \pi y}{b} \mathbf{k} \tag{9.10}
\]
\[
\mathbf{H}(\mathbf{r}) = \frac{An \pi}{\omega \mu_0 b} \sin \kappa_x x \cos \frac{n \pi y}{b} \mathbf{i} - \frac{A \kappa_x}{\omega \mu_0} \cos \kappa_x x \sin \frac{n \pi y}{b} \mathbf{j}
\]

\(^{45}\)For a derivation of the magnetic field for a wave propagating in the \( z \) direction and a discussion on wave guides in general, see The Theory of the Electromagnetic Field by David M. Cook (Prentice-Hall, Englewood Cliffs, NJ, 1975) or Introduction to Electrodynamics by David J. Griffiths (Prentice-Hall, Upper Saddle River, NJ, 1999), Third Edition. The first of these books, out of print since the early 1990’s, became available in a Dover reprint in January, 2003, but is now (January 2017) also out of print. It may still be available in your local library.
To simplify these equations further, we divide $E$ by $A$ and $H$ by $An\pi/\omega \mu_0 b$, and we elect to measure lengths in units of $b$ by introducing the variables $\overline{y} = y/b$ and $\overline{x} = x/b$. Equation (9.10) then becomes

$$\begin{align*}
E(\mathbf{r}) &= \frac{E(\mathbf{r})}{A} = \cos \kappa_x b \overline{x} \sin n\pi \overline{y} \hat{k} \\
H(\mathbf{r}) &= \frac{H(\mathbf{r})}{An\pi/\omega \mu_0 b} = \sin \kappa_x b \overline{x} \cos n\pi \overline{y} \hat{i} - \frac{\kappa_x b}{n\pi} \cos \kappa_x b \overline{x} \sin n\pi \overline{y} \hat{j}
\end{align*}$$

and we find that the field of interest depends on two parameters $n$ and $\kappa_x b$. Further, Eq. (9.9) imposes the constraint

$$\left(\frac{\kappa_x b}{n\pi}\right)^2 = \left(\frac{\omega b}{n\pi c}\right)^2 - 1 \quad (9.12)$$

on these parameters. The only non-zero components of the fields in Eq. (9.11) are

$$\begin{align*}
\overline{E}_z &= \cos \kappa_x b \overline{x} \sin n\pi \overline{y} \\
\overline{H}_x &= \sin \kappa_x b \overline{x} \cos n\pi \overline{y} \\
\overline{H}_y &= -\frac{\kappa_x b}{n\pi} \cos \kappa_x b \overline{x} \sin n\pi \overline{y}
\end{align*}$$

(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

$$\begin{align*}
\overline{E}_z &= \cos 2\pi \overline{x} \sin 2\pi \overline{y} \\
\overline{H}_x &= \sin 2\pi \overline{x} \cos 2\pi \overline{y} \\
\overline{H}_y &= -\cos 2\pi \overline{x} \sin 2\pi \overline{y}
\end{align*}$$

(9.14)

where $\overline{x}$ can range over any values—we choose $0 \leq \overline{x} \leq 1$—but, to be inside the guide, $\overline{y}$ is confined to the region $0 \leq \overline{y} \leq 1$. Each component of these fields can now be represented by a two-dimensional array. The $H$ field, which has two non-zero components, then is translated into two such arrays; the $E$ field, which has only one non-zero component, requires only one such array. To be explicit, we determine values of $\overline{H}_x(\overline{x}, \overline{y}), \overline{H}_y(\overline{x}, \overline{y}),$ and $\overline{E}_z(\overline{x}, \overline{y})$ over the region $0 \leq \overline{x}, \overline{y} \leq 1$, dividing each axis into 30 segments of length 0.1333, which will entail evaluating the field components at 30 values of $\overline{x}$ and 30 values of $\overline{y}$—a total of $30 \times 30 = 900$ values. Normally, these field components would be stored in three $30 \times 30$ two-dimensional arrays. Because of our declared file format, however, we must view ourselves as needing storage for three $30 \times 30 \times 1$ three-dimensional arrays. We shall, however, view the structure as a single $30 \times 30 \times 1 \times 3$ three-dimensional array, $H$, with $H(\ast,\ast,1,1)$ storing $\overline{H}_x$, $H(\ast,\ast,1,2)$ storing $\overline{H}_y$, and $H(\ast,\ast,1,3)$ storing $\overline{E}_z$.$^{46}$

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$^{47,48}$

---

$^{46}$We have supposed indices starting at 1. In languages where indices start at 0, the associations would, of course, be $H(\ast,\ast,0,0)$ storing $\overline{H}_x$, $H(\ast,\ast,0,1)$ storing $\overline{H}_y$, and $H(\ast,\ast,0,2)$ storing $\overline{E}_z$.

$^{47}$Note that the third index of $H$ is always one, the only value when the dimension of the array in that index is 1.

$^{48}$Note also that, for computational efficiency, various quantities involving only $\mathbf{F}$ have been calculated in the outer loop.
9.14. CREATING AND STORING TWO-DIMENSIONAL VECTOR ARRAYS

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

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
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 9.11. 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.49 Once the program has been executed, the file wavegd.f.dat will exist in the default directory.

49 For instructions on how to compile and run the program, see Section 9.12.1 and the Local Guide.

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 9.12, the main routine continues by setting appropriate values for a few other quantities with the lines

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

Then we evaluate the field components in Eq. (9.14) with the lines listed in Table 9.13.50,51,52

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 9.14. A full listing of this program will be found in Section 9.H.

50 Note that the third index of H is always zero, 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 Note, finally, the use of braces to create blocks of statements that the compiler is to see as single statements.
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.\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 `wavegd_c.dat` will exist in the default directory.

## 9.14.6 ... with PYTHON

A PYTHON program for creating and storing the necessary arrays is similar to the program discussed in Section 9.12.6. After identifying the program in a comment, including needed modules, and defining assorted constants with the lines
Table 9.12: Opening lines of program wavegd.c.

```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 */

    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 */
        }
    }
}
```

Table 9.13: 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 */
    }
}
```

```
# wavegd.py

import numpy as np # Import needed module.
narr = 3 # Set number of arrays.
xdim = 30; ydim = 30; zdim = 1 # Set dimensions: 30x30x1.
twopi = 2.0*np.pi # Set constant.
dx = twopi/(xdim-1.0) # Set increments.
dy = dx
H=np.zeros((xdim,ydim,zdim,narr)) # Prepare for values
```
Table 9.14: Closing lines of program wavegd.c.

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

fprintf( fptr, "Waveguide; Author: David M. Cook; Date: 15 June 1995\n" );
fprintf( fptr, "H and E fields in rectangular waveguide\n" );
fprintf( fptr, "Program described in CPSUP\n" );
fprintf( fptr, "**\n" );
fprintf( fptr, "**\n" );
fprintf( fptr, "%d\n", narr );
fprintf( fptr, "%d\n", xdim );
fprintf( fptr, "%d\n", ydim );
fprintf( fptr, "%d\n", zdim );
for( n=0; n<narr; n++ )
{
    for( y=0; y<ydim; y++ )
    {
        for( x=0; x<xdim; x++ )
        {
            fprintf( fptr, "%e\n", H[x][y][0][n] );
        }
    }
}
fclose( fptr );
```

we evaluate the field components in Eq. (9.14) with the lines

```python
for ix in np.arange(xdim):
    xf = ix*dx; sxf = np.sin(xf); cxf = np.cos(xf)
for iy in np.arange(ydim):
    yf = iy*dy; syf = np.sin(yf); cyf = np.cos(yf)
    H[ix,iy,0,0] = sxf * cyf
    H[ix,iy,0,1] = -cxf * syf
    H[ix,iy,0,2] = -H[ix,iy,0,1]
```

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 9.15. A full listing of this program will be found in Appendix 9.I.

The program itself is stored in the file `wavegd.py` in the directory `$HEAD/python`. It can be copied into your own directory and executed or—should this be necessary for your application—edited before being executed to create the desired file. Once the program has been executed, the file `wavegd_python.dat` will exist in the default directory.

---

54 Note that the third index of H is always zero, the only value when the dimension of the array in that index is 1.
55 Note also that, for computational efficiency, various quantities involving only xf have been calculated in the outer loop.
56 For instructions on how to run the program, see Section 9.12.6 and the Local Guide.
Table 9.15: Closing lines of program wavegd.py.

```python
f = open( 'wavegd_python.dat', 'w' )
f.write( 'Waveguide; Author: David M. Cook; Date: 31 August 2018
' );
f.write( 'H and E fields in rectangular waveguide
' );
f.write( '**
' );
f.write( '**
' );
f.write( str(narr)+'
' )
f.write( str(xdim)+'
' )
f.write( str(ydim)+'
' )
f.write( str(zdim)+'
' )
for n in np.arange(narr):
    for iy in np.arange(ydim):
        for ix in np.arange(xdim):
            f.write( str(H[ix,iy,0,n])+'
' )
f.close()
```

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

\[
\frac{[E_x(x, y, z), E_y(x, y, z), E_z(x, y, z)]}{q/4\pi\varepsilon_0 a^2}
= \left[ \begin{array}{c}
\frac{[x-1, y-1, z]}{[(x-1)^2 + (y-1)^2 + z^2]^{3/2}} - \frac{[x+1, y-1, z]}{[(x+1)^2 + (y-1)^2 + z^2]^{3/2}} \\
+ \frac{[x+1, y+1, z]}{[(x+1)^2 + (y+1)^2 + z^2]^{3/2}} - \frac{[x-1, y+1, z]}{[(x-1)^2 + (y+1)^2 + z^2]^{3/2}}
\end{array} \right]
\]  

(9.15)

To be explicit, we determine values of the field components over the region \(-2 \leq x, y, z \leq 2\), 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 \(E_z\).\(^{57}\)

\(^{57}\)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 \(E_z\).
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.J, is named \texttt{quadpole.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 \(30 \times 30 \times 30 \times 3\) array needed for the three components of the field and evaluating several useful constants. Since three arrays are to be stored, the array \(E\) for the electric field components is dimensioned at \(E(30,30,30,3)\), with \(E_x\) stored in \(E(*,*,*,1)\), \(E_y\) stored in \(E(*,*,*,2)\), and \(E_z\) stored in \(E(*,*,*,3)\).
- 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 \(XFM, XFP, YFM\) and \(YFP\) for \(x−1, x+1, y−1,\) and \(y+1\), respectively; variables \(XFMS, XFPS, YFMS, YFPS\), and \(ZFS\) for \((x−1)^2, (x+1)^2, (y−1)^2, (y+1)^2,\) and \(z^2\), 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.

As with previous programs, \texttt{quadpole.f} can be copied from the directory \texttt{$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{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.K, is named \texttt{quadpole.c} and contains sections
• Identifying the program, including standard libraries, defining constants, and declaring necessary variables. To facilitate translating integers x, y, and z into the corresponding real coordinates xf, yf, and zf, we introduce symbols \([x\text{range},y\text{range},z\text{range}]\) 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 \(E[30][30][30]\), with \(E_x\) stored in \(E[*][*][0]\), \(E_y\) stored in \(E[*][*][1]\), and \(E_z\) stored in \(E[*][*][2]\). Finally, we introduce variables xf, xfp, yfm and yfp for \(x-1, x+1, y-1,\) and \(y+1\), respectively; variables xfms, xfps, yfms, yfps, and zfs for \((x-1)^2, (x+1)^2, (y-1)^2,\) and \(z^2\), respectively; and variables rmm, rpm, 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 \([x0,y0,z0]+[x\text{range},y\text{range},z\text{range}]\). 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.

• Opening the file, writing the heading lines and the field components into the file, and closing the file.

As with previous programs, quadpole.c can be copied from the directory \$HEAD/cc into your own directory, compiled, linked, and run or—should this be necessary for your application—edited before being compiled, linked, and run to create the desired file.\(^59\) Once the program has been executed, the file quadpole.c.dat will exist in the default directory.

9.15.6 ... with PYTHON

A PYTHON program for creating and storing three-dimensional vector arrays is similar to the other PYTHON programs presented so far, in particular the two-dimensional scalar program in Section 9.12.6. 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 Appendix 9.L, is named quadpole.py and contains sections

• Identifying the program, importing necessary modules, and setting values for several constants. Since three arrays are to be stored, the array \(E\) for the electric field components is dimensioned at \(E[30,30,30,3]\), with \(E_x\) stored in \(E[*,*,*,*][0]\), \(E_y\) stored in \(E[*,*,*,*][1]\), and \(E_z\) stored in \(E[*,*,*,*][2]\). Finally, we introduce variables xf, xfp, yfm and yfp for \(x-1, x+1, y-1,\) and \(y+1\), respectively; variables xfms, xfps, yfms, yfps, and zfs for \((x-1)^2, (x+1)^2, (y-1)^2,\) and \(z^2\), respectively; and variables rmm, rpm, 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 \([x,y,z]0\) to its final value \([x,y,z]0+[x,y,z]\text{range}\). 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.

• Opening the file, writing the heading lines and the field components into the file, and closing the file.

As with previous programs, quadpole.py can be copied from the directory \$HEAD/python into your own directory, executed or—should this be necessary for your application—edited before being executed to create the desired file.\(^60\) Once the program has been executed, the file quadpole.python.dat will exist in the default directory.

\(^{59}\) For instructions on how to compile and run the program, see Section 9.12.2 and the Local Guide.

\(^{60}\) For instructions on how to run the program, see Section 9.12.6 and the Local Guide.
9.16 Reading Files

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

9.16.1 ... 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 `irrad_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

```fortran
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

```fortran
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

```fortran
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

```fortran
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 `irrad_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.M.

---

61 Reading the dimensions of the necessary array from the file at execution time is simple enough but creating the array at that time is much trickier. We choose not to address the necessary strategy in this book.
9.16. READING FILES

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.\(^{62}\) 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;
fscanf( 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.N.

---

\(^{62}\)Reading the dimensions of the necessary array from the file at execution time is simple enough but creating the array at that time is much trickier. We choose not to address the necessary strategy in this book.
9.16.6 ... with PYTHON

Data can be read from the files in PYTHON with a variety of statements. Unfortunately, the structure of the file must be known before we can create a program to read it. In this subsection, then, we will develop a PYTHON program to read the file `irrad_python.dat` created in Section 9.12.6, 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 the statement

```python
f = open('irrad_python.dat', 'r')
```

Then we must read past the five lines of comments with statements like\(^{53}\)

```python
ln1=f.readline(); ln2=f.readline(); ln3=f.readline()
ln4=f.readline(); ln5=f.readline()
```

Next, we read the four dimensions from the file with the statements

```python
narr=int(f.readline())
ix=int(f.readline()); iy=int(f.readline()); iz=int(f.readline())
```

Here, the action of the command `int` both strips off the end-of-line character and converts the string read from the file into an integer. (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 initialize an appropriate array for the data, read the values one at a time from the file, and close the file with the statements

```python
import numpy as np
I = np.zeros((ix,iy))
for j in np.arange(iy):
    for i in np.arange(ix):
        I[i,j] = float(f.readline())
f.close()
```

Here, the command `float` both strips the end-of-line character from what is read and converts the result from a character string into a floating point number. We also recognize that the file really contains only a $50 \times 50$ array even though our construction of the file envisioned it to be an equivalent $50 \times 50 \times 1 \times 1$ array, so we have created $I$ in the more simple form.\(^{64}\)

Execution of these statements will recreate the variable $I$ as it was when the file `irrad_python.dat` was created in Section 9.12.6. A fully commented and properly ordered listing of the program `read_irrad.py` is presented in Appendix 9.O.

\(^{53}\)Remember that the character `\n` will at the end of each line will be included in the values assigned to the variables `ln?`.

\(^{64}\)Indeed, creating it as an array with two dimensions equal to 1 results in an error message when we attempt to invoke a loop that is iterated only once.
9.17 References


For PYTHON, see Section 5.17.

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 9.16 for determining the parking fee for a particular patron more explicit by expanding the two statements in the IF-THEN-ELSE structure. *Hint:* Work through several numeric examples by hand, noting particularly all decisions that you must make in order to know what arithmetic to do. *Optional:* Describe a procedure for determining the time in the lot from the dates and times of entry and exit. Assume first that the two dates are in the same month, but then give some thought to generalizing your procedure to handle cases where the two dates span two or more months or years.

9.2. Figure 9.5 shows three different alternative structures. Express each structure using (a) only CASE structures and (b) only IF-THEN-ELSE structures. In these figures, T, F, C, and B stand for true, false, condition, and block of statements, respectively. Use proper indentation as illustrated in the examples.

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

9.4. For the game of bowling, identify appropriate elementary action statements and then write an algorithm that will accept the number of pins knocked over with each ball and report the frame-by-frame scores.
9.5. (a) Cast Algorithm (6) of Section 9.2 in a form more specific to finding the largest integer in a list of integers.

(b) The algorithm shown in Table 9.17 is an alternative to the algorithm deduced in (a). Essentially, an alternative method of initializing $\text{LARGEST\%}$ is adopted. Write a few sentences identifying the advantages and disadvantages of the two different methods.

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

9.7. Basing your work on Algorithm (6) of Section 9.2, write an algorithm that will obtain words one at a time and ultimately report (1) the word that would appear last if the list were alphabetized, (2) the word that would appear first if the list were alphabetized, (3) the total number of words given, and (4) the position of each extreme word in the original list. Only one pass through the list is permitted.

9.8. Write an algorithm that will find and report all triplets of positive integers (zero excluded) $A\%$, $B\%$, $C\%$ satisfying $A\% \times A\% + B\% \times B\% = C\% \times C\%$, subject to the restriction that $A\%$, $B\%$, and $C\%$ shall all be smaller than some value $\text{MAXNUM\%}$ supplied as input. Hint: Systematically examine all possibilities, but do so thoughtfully. For example, there is no point in examining cases for which $C\% \leq A\%$ or $C\% \leq B\%$. Express your loops so that these cases (and any others that you can reject a priori) will not even be considered. Optional: For $\text{MAXNUM\%} = 20\%$, determine the number of executions of your
Table 9.17: Algorithm for Exercise 9.5.

\[
\text{SENTINEL}\% \leftarrow \langle \text{agreed-upon special value} \rangle \\
\text{LARGEST}\% \leftarrow \langle \text{assumed fictitious integer known to occur before any possible real} \\
\text{integer in the list} \rangle \\
\text{LOOP} \\
\text{READ ITEM}\% \\
\text{EXIT_LOOP WHEN ITEM}\% = \text{SENTINEL}\% \\
\text{IF ITEM}\% > \text{LARGEST}\% \\
\text{THEN LARGEST}\% \leftarrow \text{ITEM}\% \\
\text{END_IF} \\
\text{END_LOOP} \\
\text{Report LARGEST}\%
\]

Table 9.18: Procedure for Exercise 9.9.

\[
\text{PROCEDURE} \ \ast\ast\ast\ast\ast\ast \\
\text{SCANEND}\% \leftarrow N\% \\
\text{LOOP} \\
\text{CARD}\% \leftarrow 1\% \\
\text{Obtain word on card CARD}\% \text{ and store in WORD}$ \\
\text{LATEST\_WORD}$\% \leftarrow \text{WORD}$ \\
\text{LATEST\_CARD}\% \leftarrow \text{CARD}\% \\
\text{LOOP} \\
\text{CARD}\% \leftarrow \text{CARD}\% + 1\% \\
\text{Obtain word on card CARD}\% \text{ and store in WORD}$ \\
\text{IF WORD}$ occurs after LATEST\_WORD}$ \\
\text{THEN BEGIN_BLOCK} \\
\text{LATEST\_WORD}$\% \leftarrow \text{WORD}$ \\
\text{LATEST\_CARD}\% \leftarrow \text{CARD}\% \\
\text{END_BLOCK} \\
\text{END_IF} \\
\text{EXIT_LOOP WHEN CARD}\% = \text{SCANEND}\% \\
\text{END_LOOP} \\
\text{Exchange card LATEST\_CARD}\% \text{ with card SCANEND}\% \\
\text{SCANEND}\% \leftarrow \text{SCANEND}\% - 1\% \\
\text{EXIT_LOOP WHEN SCANEND}\% = 1\% \\
\text{END_LOOP} \\
\text{END_PROCEDURE}
\]

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 9.18 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 9.19 and choose a suitable name for the procedure.

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
        ENDLOOP
    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
        ENDLOOP
    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

9.11. Starting with the approximation

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

deduce the finite difference approximation

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

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

9.13. (a) Copy `laplace_file.f` from $\text{HEAD/fortran}$ 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 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. (a) Copy `laplace_file.c` from $\text{HEAD/cc}$ to your directory, naming it `laplace15_file.c`. Then compile, link, and run `laplace15_file.f` to generate the file `laplace.c.dat`, which you should rename `laplace15.c.dat`.

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

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

(d) Import both `laplace15.c.dat` and `laplace30.c.dat` into an available program for graphical visualization and then
   i. generate a graphical display of each solution, either a contour map in the $xy$ plane or a surface plot over the $xy$ plane (or perhaps both). In either case (or both cases), make sure the axes are labeled correctly with the proper coordinate values. Warning: This latter requirement is a bit subtle. Beware.
   ii. develop a way to compare the two solutions at those grid points that are common to the two and display the differences graphically.

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

(f) Edit the last program again so that iteration is stopped when the maximum change falls below a tolerance that is specified as input when the program is run (or—to prevent infinite loops—when the number of iterations exceeds some maximum value). Arrange for the program to display the number of iterations carried out when the solution finally converges. Compile, link, and run this last program and explore the way the number of iterations varies with the tolerance specified. Was 300 iterations as a trial in the original programs vast overkill?
9.15. Consider two circular current loops, each of radius $a$ and lying with its center on and its plane perpendicular to the $z$ axis. The first loop is centered at the point $(0,0,b)$ and the second loop is centered at the point $(0,0,-b)$. The axial component of the magnetic field at the point $(0,0,z)$ is given by the equation

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

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 as a two-dimensional scalar function of $z/a$ and $b/a$. Suggestion: Write values of $B/B_0$ into the file.

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

$$\mathbf{r} = x(t) \hat{i} + y(t) \hat{j} + z(t) \hat{k}$$

You desire to fathom out the general character of this trajectory by using a graphical visualization tool that does not have much computational capability. Thus, you must generate the data using one tool but will visualize the trajectory with another tool. You elect to use an ASCII file to communicate the data from the first tool to the second. Suppose that the ASCII file produced by the first tool is to be structured as follows:

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

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

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

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

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

$$p_{3,1,0}(x,y,z) = \frac{8}{(2\pi)^2} \rho \left[ \left( 1 - \frac{\rho}{\rho_0} \right)^2 e^{-2\rho/\rho_0} \right]$$

$$p_{3,1,1}(x,y,z) = \frac{4}{(2\pi)^2} \rho^2 \left[ \left( 1 - \frac{\rho}{\rho_0} \right)^2 e^{-2\rho/\rho_0} \right]$$

$$p_{3,2,1}(x,y,z) = \frac{3}{(2\pi)^2} \rho^4 e^{-2\rho/\rho_0} \cos^2 \theta (1 - \cos^2 \theta)$$

$$p_{3,2,2}(x,y,z) = \frac{3}{(2\pi)^2} \rho^4 e^{-2\rho/\rho_0} (1 - \cos^2 \theta)^2$$

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

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

$$p(x,y,z,t) = A \sin \frac{\pi x}{a} \sin \frac{\pi y}{a} \sin \frac{\pi z}{a} \cos \omega t$$
where \( l, m, \) and \( n \) are positive integers. Using at least one language, create files containing the pressure distribution inside the box at \( t = 0 \) for several different values of \( l, m, \) and \( n \).

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

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

Expressing coordinates in terms of the dimensionless variables \( \bar{x} = x/a \) and \( \bar{y} = y/a \) and measuring \( \mathbf{E} \) in the unit \( q/(4\pi\epsilon_0a^2) \), create a file conforming to the structure described in Section 9.3.2 and containing the \( x \) and \( y \) components of this field over the interval \(-2a \leq x, y \leq 2a\). Divide the interval in each coordinate direction into about 25 segments, but choose the precise number carefully so as to avoid division by zero in evaluating the field at any point. Optional: Read your file into a suitable graphical display program and produce graphs showing the character of this field.

9.20. The velocity field of an object rotating about the \( z \) axis with angular momentum \( \omega \) is given in terms of the angular velocity and the position vector \( \mathbf{r} \) by the expression

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

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

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

Expressing coordinates in terms of the dimensionless variables \( \bar{x} = x/a \), \( \bar{y} = y/a \), and \( \bar{z} = z/a \) and measuring \( \mathbf{E} \) in the unit \( q/(4\pi\epsilon_0a^2) \), 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 \( 43^\circ 40' \text{N}, 73^\circ 45' \text{W} \); Grand Junction, CO, is at \( 39^\circ 5' \text{N}, 108^\circ 33' \text{W} \); Los Angeles, CA, is at \( 34^\circ 3' \text{N}, 118^\circ 15' \text{W} \); Appleton, WI, is at \( 44^\circ 16' \text{N}, 88^\circ 25' \text{W} \); Calcutta India, is at \( 22^\circ 32' \text{N}, 88^\circ 20' \text{E} \); Sydney, Australia, is at \( 33^\circ 52' \text{S}, 151^\circ 12' \text{E} \); Paris, France, is at \( 48^\circ 49' \text{N}, 2^\circ 29' \text{E} \); and Stockholm, Sweden, is at \( 59^\circ 21' \text{N}, 18^\circ 4' \text{E} \).
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, "\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<yd; y++ )
{
    for( x=0; x<x; x++ )
    {
        fprintf( fptr, "%e\n", I[x][y] );
    }
}

/***** CLOSE FILE *****/
fclose( fptr );
9.C Listing of irrad.py

# irrad.py

import numpy as np  # Import needed package
narr = 1; ix = 50; iy = 50; iz = 1  # Set dimensions
I = np.zeros((ix,iy));  # Prepare array for irradiance
threepi = 3.0*np.pi;  # Set constant
dx = 2.0*threepi/(ix-1.0);  # Set increments
dy = dx

for j in np.arange(iy):  # Evaluate irradience
    yf = j*dy - threepi
    for i in np.arange(ix):
        xf = i*dx - threepi
        I[i,j] = (np.sin(xf)/xf)**2 * (np.sin(yf)/yf)**2

# ***** Open file for writing and write initial lines *****

f = open( 'irrad_python.dat', 'w'
)
f.write( 'Irradiance; Author: David M. Cook; Date: 30 August 2018
')
f.write( 'Fraunhofer diffraction at square aperture
')
f.write( 'Program described in CPSUP
')
f.write( '***
')
f.write( '***
')
f.write( str(narr)+'
')
f.write( str(ix)+'
')
f.write( str(iy)+'
')
f.write( str(iz)+'
')

# ***** Write irradience to file, one value per line *****

for j in np.arange(iy):
    for i in np.arange(ix):
        f.write( str( I[i,j] ) + '\n' )

# ***** Close file *****

f.close()
9.D 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
PFACTOR = 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
9. E 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;
            P[xs][ys][zs] = P[xs][ys][zs] + pfactor;
        }
    }
}
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.F Listing of pdens.py

# pdens.py

# ***** Preliminaries *****

import numpy as np  # Import needed package
narr = 1; xdim = 30  # Set dimensions
ydim = 30; zdim = 30
twopi = 2.0*np.pi  # Set constant
xrange=20; yrange=20; zrange=20  # Set lengths of intervals
x0=-10.0; y0=-10.0; z0=-10.0  # Set starting values of x,y,z
dx = xrange/(xdim-1.0)  # Set increments
dy = yrange/(ydim-1.0)
dz = zrange/(zdim-1.0)
pfactor = 1.0/(twopi*27.0**3)  # Evaluate constant
P = np.zeros( (xdim,ydim,zdim) )  # Prepare array for values

# ***** Evaluate probability density *****

for iz in np.arange(zdim):
zf = iz * dz + z0
zs = zf**2
    for iy in np.arange(ydim):
yf = iy * dy + y0
ys = yf**2
        for ix in np.arange(xdim):
xf = ix * dx + x0
xs = xf**2
    rhos = xs + ys + zs
rho = np.sqrt(rhos);
P[ix,iy,iz] = pfactor * np.exp( -2.0*rho/3.0 ) * \
( 9*zs*zs - 6*zs*rhos + rhos*rhos )

# ***** Open file in write mode, get file ID *****

f = open( 'pdens_python.dat', 'w' )

# ***** Write heading lines to file *****

f.write( 'Probability density; Author: David M. Cook;' )
f.write( ' Date: 31 August 2018
' )
f.write( 'The probability density for the electron in the
' )
f.write( '(n,l,m) = (3,2,0) state in the hydrogen atom.
' )
f.write( '**
' )

f.write( str(narr)+'
' )
f.write( str(zdim)+'
' )
f.write( str(ydim)+'
' )
f.write( str(xdim)+'
' )

# ***** Write probability density to file; close file *****

f.write( str(narr)+'
' )
f.write( str(zdim)+'
' )
f.write( str(ydim)+'
' )
f.write( str(xdim)+'
' )
for iz in np.arange(zdim):
    for iy in np.arange(ydim):
        for ix in np.arange(xdim):
            f.write( str(P[ix,iy,iz]) + '\n' )

f.close()
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 );
9.1 Listing of wavegd.py

# wavegd.py

# ***** Preliminaries ******

import numpy as np # Import needed package
narr = 3 # Set number of arrays
xdim = 30; ydim = 30; zdim = 1 # Set dimensions: 30x30x1
twopi = 2.0*np.pi # Set constant
dx = twopi/(xdim-1.0) # Set increments
dy = dx
H=np.zeros((xdim,ydim,zdim,narr)) # Prepare for values

# ***** Evaluate field components ******

for ix in np.arange(xdim):
    xf = ix*dx; sxf = np.sin(xf); cxf = np.cos(xf)
    for iy in np.arange(ydim):
        yf = iy*dy; syf = np.sin(yf); cyf = np.cos(yf)
        H[ix,iy,0,0] = sxf * cyf
        H[ix,iy,0,1] = -cxf * syf
        H[ix,iy,0,2] = -H[ix,iy,0,1]

# ***** Open file in write mode, get file ID ******

f = open( 'wavegd_python.dat', 'w' )

# ***** Write heading lines to file ******

f.write( 'Waveguide; Author: David M. Cook; Date: 31 August 2018
' );
f.write( 'H and E fields in rectangular waveguide
' );
f.write( 'Program described in CPSUP
' );
f.write( '**
' );
f.write( '---
' );
f.write( str(narr)+'
' )
f.write( str(xdim)+'
' )
f.write( str(ydim)+'
' )
f.write( str(zdim)+'
' )

# ***** Write field to file; close file ******

for n in np.arange(narr):
    for iy in np.arange(ydim):
        for ix in np.arange(xdim):
            f.write( str(H[ix,iy,0,n])+'
' )

f.close()
9.J 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
        YFPS = YFP*YFP
        DO I = 1, IX
            XF = (I-1)*DX + X0
            XFM = XF - 1.0
            XFPS = XFP*XFP
            RMM = (XFM + YFM + ZFS)**1.5
            RMP = (XFM + YFPS + ZFS)**1.5
            RPP = (XFPS + YFPS + ZFS)**1.5
            RPM = (XFPS + YFM + ZFS)**1.5
            E(I,J,K,1) = XFM * (1.0/RMM - 1.0/RMP) ! Ex
                          + XFP * (1.0/RPP - 1.0/RPM)
            E(I,J,K,2) = YFM * (1.0/RMM - 1.0/RMP) ! Ey
                          + YFP * (1.0/RPP - 1.0/RPM)
            E(I,J,K,3) = ZF * (1.0/RMM - 1.0/RMP) ! Ez
                          + 1.0/RPP - 1.0/RPM)
        END DO
    END DO
END DO

! OPEN FILE FOR WRITING *****

OPEN( UNIT = 1, FILE = 'quadpole.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.K Listing of quadpole.c

/* PROGRAM quadpole.c */

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

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

main()
{
***** 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, rmp, rpm, rpp; /* For denominators */
int n, x, y, z; /* For loop control */
dx = xrange/(xdim - 1.0); /* Set increments */
dy = yrange/(ydim - 1.0);
dz = zrange/(zdim - 1.0);

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

for( z=0; z<zdim; z++ )
{
    zf = z * dz + z0;
zfs = zf*zf;
    for( y=0; y<ydim; y++ )
    {
        yf = y * dy + y0;
yfm = yf - 1.0;
yfp = yf + 1.0;
yfms = yfm*yfm;
yfps = yfp*yfp;
        for( x=0; x<xdim; x++ )
        {
            xf = x * dx + x0;
xfm = xf - 1.0;
xfp = xf + 1.0;
xfms = xfm*xfm;
        }
    }
}
xfps = xfp*xfp;
rm = pow(xfms + yfms + zfs, 1.5);
rmp = pow(xfms + yfps + zfs, 1.5);
rpp = pow(xfps + yfms + zfs, 1.5);
rpm = pow(xfps + yfps + 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/rmp) /* Ey */
    + yfp*(1.0/rpp - 1.0/rpm);
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(fptr, "Quadrapole; Author: Chris Schmidt;"
    );
fprintf(fptr, " Date: 27 June 1995\n");
fprintf(fptr, "The electric field from 4 point charges positioned\n");
fprintf(fptr, "on the corners of a square.\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 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( fptr, "%e\n", E[x][y][z][n] );
            }
        }
    }

    /* CLOSE FILE *****/
    fclose( fptr );
}

# quadpole.py

import numpy as np
narr = 3
xrange = 4.0
yrange = 4.0
zrange = 4.0
x0 = -2.0; y0 = -2.0; z0 = -2.0
E = np.zeros((xdim, ydim, zdim, narr))

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

# ***** Evaluate electric field *****

for z in np.arange(zdim):
zf = z * dz + z0; zfs = zf ** 2
for y in np.arange(ydim):
yf = y * dy + y0; yfm = yf - 1.0; yfp = yf + 1.0
yfms = yfm ** 2; yfps = yfp ** 2
for x in np.arange(xdim):
xf = x * dx + x0; xfm = xf - 1.0; xfp = xf + 1.0
xfms = xfm ** 2; xfps = xfp ** 2
rmm = (xfms + yfms + zfs) ** 1.5
rpm = (xfms + yfms + zfs) ** 1.5
rpp = (xfps + yfps + zfs) ** 1.5
rpm = (xfps + yfms + zfs) ** 1.5
E[x, y, z, 0] = xfm * (1.0 / rmm - 1.0 / rpm) \
+ xfp * (1.0 / rpp - 1.0 / rpm)
E[x, y, z, 1] = yfm * (1.0 / rmm - 1.0 / rpm) \
+ yfp * (1.0 / rpp - 1.0 / rpm)
E[x, y, z, 2] = zf * (1.0 / rmm - 1.0 / rmp \
+ 1.0 / rpp - 1.0 / rpm)

# ***** Open file in write mode; get ID *****

f = open( 'quadpole_python.dat', 'w' )

# ***** Write heading lines to file *****

f.write( 'Quadrapole; Author: David M. Cook;' )
f.write( ' Date: 31August 2018\n' )
f.write( 'The electric field from 4 point charges positioned\n' )
f.write( 'on the corners of a square.\n' )
f.write( 'Program described in CPSUP\n' )
f.write( '**\n' )

f.write( str(narr) + '\n' )
f.write( str(zdim) + '\n' )
f.write( str(ydim) + '\n' )
f.write( str(xdim) + '\n' )
# ***** Write field components to file; close file *****

for n in np.arange(narr):
    for z in np.arange(zdim):
        for y in np.arange(ydim):
            for x in np.arange(xdim):
                f.write( str(E[x,y,z,n])+'
' )

f.close()
9.M 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.N 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<i; j++ )
        for( i=0; i<ix; i++ )
          fscanf( fptr, "%g", &I[i][j][k][n] );

  fclose( fptr ); /* Close file */

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

9.O Listing of read_irrad.py

# read_irrad.py

import numpy as np # Import needed package

# ***** Open file, read past five comment lines *****
f = open( 'irrad_python.dat', 'r' )
ln1=f.readline(); ln2=f.readline(); ln3=f.readline()
ln4=f.readline(); ln5=f.readline()

# ***** Read controlling parameters; convert to integers *****
narr=int(f.readline())
ix=int(f.readline()); iy=int(f.readline()); iz=int(f.readline())

# ***** Create for data; read data; close file *****
I = np.zeros( (ix,iy) )
for j in np.arange(iy):
    for i in np.arange(ix):
        I[i,j] = float( f.readline() )
f.close()
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: \textit{Numerical Recipes: The Art of Scientific Computing} (hereafter referred to as the main source book) and \textit{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 $\texttt{NRHEAD}$.\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 $\texttt{NRHEAD}/\texttt{recipes\_language}/\texttt{recipes}$ 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.
Figure 10.1: The directory structure for the FORTRAN-77 and C files in the second edition of Numerical Recipes.

- 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

   \texttt{cp \$NRHEAD/recipes\_f/demo/src/xflmoon.f .}

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

   \texttt{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

   \texttt{f77 \(-o xflmoon.xf xflmoon.f flmoon.f\)}

   If this statement generates no error messages, then no further recipes are needed.\textsuperscript{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 \texttt{julday} and \texttt{caldat} will be identified, so we need the statements

   \texttt{cp \$NRHEAD/recipes\_f/recipes/caldat.f .}
   \texttt{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 \texttt{xflmoon.f} and the recipe \texttt{flmoon.f}, noting the statements that invoke subroutines \texttt{flmoon} and \texttt{caldat} and function \texttt{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

   \texttt{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 \texttt{f77} 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{9}

   \texttt{./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.

\textsuperscript{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.

\textsuperscript{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 \texttt{f90} and \texttt{f95}, respectively.

\textsuperscript{8}The routine \texttt{xflmoon.f} invokes the FORTRAN \texttt{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.

\textsuperscript{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

   \texttt{cp \$NRHEAD/recipes_c-ansi/demo/src/xflmoon.c .}

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

   \texttt{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

   \texttt{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

   \texttt{cp \$NRHEAD/recipes_c-ansi/include/nr.h .}

   Then, a repeat compilation with the statement

   \texttt{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

   \texttt{cp \$NRHEAD/recipes_c-ansi/recipes/julday.c .}

   \texttt{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

   \texttt{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 \texttt{-lm}.

---

\textsuperscript{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.

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

4. We compile and load the program with the statement\textsuperscript{12}

```
cc -o xflmoon.xc xflmoon.c flmoon.c julday.c caldat.c nrutil.c -lm
```

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

5. We run the program with the statement\textsuperscript{13}

```
./xflmoon.xc
```

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

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

\section*{10.4 Exercises}

\textbf{10.1.} The recipe `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 `xjulday.f`, the primary recipe `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 `DATES.DAT` needed by the program can be copied from the directory `$NRHEAD/recipes/f/demo/results`.

\textbf{10.2.} The recipe `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 `xavevar.f`, the primary recipe `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 coompileation.

\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).
CHAPTER 11. SOLVING ORDINARY DIFFERENTIAL EQUATIONS

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

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

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

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

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

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

or, in component form, to

\[
\begin{align*}
m \frac{d^2 x}{dt^2} &= -b \frac{dx}{dt} ; \\
m \frac{d^2 y}{dt^2} &= -b \frac{dy}{dt} ; \\
m \frac{d^2 z}{dt^2} &= -mg - b \frac{dz}{dt}
\end{align*}
\]

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

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

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

or, in component form, to

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

\end{align*}
\]

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

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

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

To cast these equations in a dimensionless form, we would start by choosing a unit of length, say \( a \)—which might be chosen arbitrarily, chosen to be one of the initial positions, or merely symbolized initially and chosen later to simplify the equations as we converge on their dimensionless form. Then, dividing by \( m \) and rescaling the dimensional coordinates by introducing the dimensionless
coordinates \( \bar{\mathbf{x}} = x/a, \bar{\mathbf{y}} = y/a, \) and \( \bar{\mathbf{z}} = z/a \) (i.e., \( \bar{\mathbf{r}} = \mathbf{r}/a \)), we find that Eqs. (11.3) and (11.5) become
\[
\frac{d^2 \bar{\mathbf{r}}}{dt^2} = -\frac{g}{a} \hat{k} - \frac{b}{m} \frac{d \bar{\mathbf{r}}}{dt}
\] (11.8)
and
\[
\frac{d^2 \bar{\mathbf{r}}}{dt^2} = -\frac{g}{a} \hat{k} - \frac{ba}{m} \frac{d \bar{\mathbf{r}}}{dt} \sqrt{\left(\frac{d\bar{\mathbf{x}}}{dt}\right)^2 + \left(\frac{d\bar{\mathbf{y}}}{dt}\right)^2 + \left(\frac{d\bar{\mathbf{z}}}{dt}\right)^2}
\] (11.9)
respectively. Next, we recognize that \( \sqrt{g/a} \) is dimensionally a frequency. Hence, the variable \( t = t \sqrt{g/a} \) provides a suitable rescaling of the independent variable. In terms of \( t \), Eqs. (11.8) and (11.9) become
\[
\frac{d^2 \bar{\mathbf{r}}}{dt^2} = -\ddot{\mathbf{r}} - \frac{b}{m} \sqrt{\frac{g}{a}} \frac{d \bar{\mathbf{r}}}{dt}
\] (11.10)
and
\[
\frac{d^2 \bar{\mathbf{r}}}{dt^2} = -\ddot{\mathbf{r}} - \frac{b}{m} \sqrt{\frac{g}{a}} \frac{d \bar{\mathbf{r}}}{dt} \sqrt{\left(\frac{d\bar{\mathbf{x}}}{dt}\right)^2 + \left(\frac{d\bar{\mathbf{y}}}{dt}\right)^2 + \left(\frac{d\bar{\mathbf{z}}}{dt}\right)^2}
\] (11.11)
Finally, we introduce the symbol \( \bar{\mathbf{J}} \) for the dimensionless quantity \( b \sqrt{a/g} \) in the first of these equations and for the dimensionless quantity \( ba/m \) in the second.\(^3\) In the end, the equations in dimensionless form are
\[
\frac{d^2 \bar{\mathbf{r}}}{dt^2} = -\ddot{\mathbf{r}} - \bar{\mathbf{J}} \frac{d \bar{\mathbf{r}}}{dt}
\] (11.12)
and
\[
\frac{d^2 \bar{\mathbf{r}}}{dt^2} = -\ddot{\mathbf{r}} - \bar{\mathbf{J}} \frac{d \bar{\mathbf{r}}}{dt} \sqrt{\left(\frac{d\bar{\mathbf{x}}}{dt}\right)^2 + \left(\frac{d\bar{\mathbf{y}}}{dt}\right)^2 + \left(\frac{d\bar{\mathbf{z}}}{dt}\right)^2}
\] (11.13)
These equations are, of course, to be solved subject to the dimensionless initial conditions
\[
\bar{\mathbf{r}}(0) = \frac{r_0}{a} ; \quad \mathbf{v}(0) = \frac{d\bar{\mathbf{r}}}{dt}(0) = \frac{d(r/a)}{d(t \sqrt{g/a})}(0) = \frac{v_0}{\sqrt{ga}}
\] (11.14)
from which we conclude that dimensionless velocities are measured in units of \( \sqrt{ga} \).

In truth, however, these particular equations are not really second-order equations. They can be reduced to first-order equations by focusing attention on the components \( [v_x, v_y, v_z] \) of the velocity as the dependent variables. In those terms, the three members of Eq. (11.4) become
\[
m \frac{dv_x}{dt} = -bv_x ; \quad m \frac{dv_y}{dt} = -bv_y ; \quad m \frac{dv_z}{dt} = -mg - bv_z
\] (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)
\(^2\)Using \([\ldots]\) to indicate “the dimensions of \( \ldots \),” we argue \([g] = \text{length/time}^2, [a] = \text{length} \implies [g/a] = \text{time}^{-2} \implies [\sqrt{g/a}] = \text{time}^{-1}.
\(^3\)Remember that \( b \) is not the same in the two instances.
\(^4\)Here (and in all subsequent cases where a chosen unit is identified but not checked), you should take a moment to verify that the identified unit has the proper dimensions.
11.1. SAMPLE PROBLEMS

Figure 11.2: A three-nucleus radioactive decay.

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

$$\frac{dA}{dt} = -k_A A; \quad \frac{dB}{dt} = k_A A - k_B B; \quad \frac{dC}{dt} = k_B B$$  \hspace{1cm} (11.19)

where $k_A$ and $k_B$ are decay constants (parameters); $A(t)$, $B(t)$, and $C(t)$ are the number of nuclei of each species present; and nucleus $C$ is assumed to be stable. These equations are linear, first-order, and homogeneous, and they have constant coefficients. They are, however, coupled, since each of the second and third of them involves two of the dependent variables. They also support a conservation law: \footnote{We shall later see that conservation laws can sometimes prove valuable in assessing the accuracy of solutions.} adding the three equations yields that

$$\frac{d}{dt}(A + B + C) = 0 \quad \Rightarrow \quad A + B + C = \text{constant}$$  \hspace{1cm} (11.20)

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

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

$$\frac{d\bar{A}}{d\bar{t}} = -\bar{A}; \quad \frac{d\bar{B}}{d\bar{t}} = \bar{A} - \bar{k} \bar{B}; \quad \frac{d\bar{C}}{d\bar{t}} = \bar{k} \bar{B}$$  \hspace{1cm} (11.21)

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

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

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

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

before the problem is fully stated.

---

6 Logistic 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.
11.1. SAMPLE PROBLEMS

To cast this equation in dimensionless form, we introduce a unit of length, say \( \ell \), rescale position with the expression \( \bar{x} = x/\ell \), introduce a unit of time, say \( \tau \), rescale the physical time with the expressions \( \bar{t} = t/\tau \), and thereby transform Eq. (11.24) to

\[
\frac{d^2 \bar{x}}{d\bar{t}^2} = -\frac{\tau^2 k}{m} \bar{x} - \frac{\tau b}{m} \frac{d\bar{x}}{d\bar{t}} + \tau^2 \frac{F(\tau \bar{t})}{m \ell} \tag{11.26}
\]

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 \( \tau \) any way we like. Clearly, the choice \( \tau^2 k/m = 1 \) or \( \tau = \sqrt{m/k} \) is judicious. With this choice, the equation of motion becomes

\[
\frac{d^2 \bar{x}}{d\bar{t}^2} = -\bar{x} - \frac{b}{\sqrt{m k}} \frac{d\bar{x}}{d\bar{t}} + \frac{F(\tau \bar{t})}{k \ell} \tag{11.27}
\]

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

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

\[
\bar{x}(0) = \frac{x_0}{\ell} ; \quad \bar{v}(0) = \frac{d\bar{x}}{d\bar{t}}(0) = \frac{d(x/\ell)}{d(t/\tau)} = \frac{v_0}{\ell/\tau} \tag{11.28}
\]

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 single second-order differential equation as a pair of first-order equations, either

\[
\frac{dx}{dt} = v ; \quad m \frac{dv}{dt} = -kx - bv + F(t) \tag{11.29}
\]

in the original dimensional form, or

\[
\frac{d\bar{x}}{d\bar{t}} = \bar{v} ; \quad \frac{d\bar{v}}{d\bar{t}} = -\bar{x} - \bar{\beta} \bar{v} + \bar{F}(\bar{t}) \tag{11.30}
\]

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

$$V(t) - L \frac{di}{dt} - \frac{q}{C} - iR = 0 \quad \Rightarrow \quad L \frac{di}{dt} + iR + \frac{q}{C} = V(t) \quad (11.31)$$

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, whereon we could easily set up electronic circuits whose behavior simulated the behavior of more expensive mechanical systems. With an analog computer, we could learn about mechanical systems by observing the real-time variation of the voltages and currents at various points in an analogous electronic circuit.
of constant $k'$. Let the (horizontal) surface on which these objects slide be frictionless, and let $x_1(t)$ and $x_2(t)$ be the displacement of each object from its equilibrium point. Then, Newton’s second law yields

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

(11.35)

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

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

(11.36)

Initial conditions such as

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

(11.37)

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

### 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
\[ m \frac{d^2 r}{dt^2} = \mathbf{F} = f(r) \hat{r} \]  
(11.38)

where \( f(r) > 0 \) corresponds to a repulsive force and \( f(r) < 0 \) corresponds to an attractive force. Two articulations of this equation are in order. A more specific expression in polar coordinates is treated in an exercise. Here, we extract its components in Cartesian coordinates \((x, y)\). Since
\[ r^2 = x^2 + y^2, \quad r = x \hat{i} + y \hat{j}, \]  
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, in which a planet of mass \( m \) orbits a central sun of mass \( M \). When \( M \gg m \), as is often the case, the sun does not move appreciably under the action of the gravitational force exerted on it by the planet. Thus, we can treat the gravitational force on the planet as originating in a fixed force center, in which case
\[ f(r) = -G \frac{mM}{r^2} = -G \frac{mM}{x^2 + y^2} \]  
(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
\[ \bar{x} = \frac{x}{\ell} ; \quad \bar{y} = \frac{y}{\ell} ; \quad \bar{r} = \frac{r}{\ell} \]  
(11.44)

With this change, Eq. (11.43) becomes
\[ \frac{d^2 \bar{x}}{dt^2} = -\frac{GM}{\ell^3} \frac{\bar{x}}{(\bar{r}^2 + \bar{y}^2)^{3/2}} ; \quad \frac{d^2 \bar{y}}{dt^2} = -\frac{GM}{\ell^3} \frac{\bar{y}}{(\bar{r}^2 + \bar{y}^2)^{3/2}} \]  
(11.45)

Finally, we introduce the dimensionless time variable \( \bar{t} = t \sqrt{GM/\ell^3} \) to find that
\[ \frac{d^2 \bar{x}}{d\bar{t}^2} = -\frac{\bar{x}}{(\bar{r}^2 + \bar{y}^2)^{3/2}} ; \quad \frac{d^2 \bar{y}}{d\bar{t}^2} = -\frac{\bar{y}}{(\bar{r}^2 + \bar{y}^2)^{3/2}} \]  
(11.46)

Interestingly, in Cartesian coordinates, all parameters have disappeared from the equations.

\[ ^9 \text{Remember, too, that the problem of a charged particle moving in the field of another charged particle is mathematically identical to the planetary problem. (See exercises.)} \]
Translation of the initial conditions into dimensionless form is easier than translation of the differential equations. The positions, of course, become

\[ \mathbf{r}(0) = \frac{x_0}{\ell} \quad ; \quad \mathbf{y}(0) = \frac{y_0}{\ell} \]

To translate the velocities, we argue that

\[ \frac{dx}{dt}(0) = \frac{d(x/\ell)}{d(t\sqrt{GM/\ell^3})}(0) = \frac{v_x0}{\sqrt{GM/\ell}} ; \quad \frac{dy}{dt}(0) = \frac{d(y/\ell)}{d(t\sqrt{GM/\ell^3})}(0) = \frac{v_y0}{\sqrt{GM/\ell}} \]

(11.47)

(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

\[ \frac{mv_{\text{circ}}^2}{r_{\text{circ}}} = \frac{GMm}{r_{\text{circ}}^2} \implies v_{\text{circ}}^2 = \frac{GM}{r_{\text{circ}}} \]

(11.49)

Translated into dimensionless form, this special relationship becomes

\[ \left( \sqrt{\frac{GM}{\ell}} v_{\text{circ}} \right)^2 = \frac{GM}{\ell v_{\text{circ}}} \implies v_{\text{circ}}^2 = \frac{1}{v_{\text{circ}}} \]

(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

\[ E = \frac{1}{2} m \left( v_x^2 + v_y^2 \right) - \frac{GMm}{\sqrt{x^2 + y^2}} = \text{constant} \]

(11.51)

though the mass \( m \) can be omitted from the expression if only the constancy of \( E \) (and not its actual value) is to be examined. Similarly, in a dimensional presentation, conservation of angular momentum yields that

\[ L = m \left( xv_y - yv_x \right) = \text{constant} \]

(11.52)

where, again, \( m \) can be omitted if only the constancy of \( L \) is to be assessed. Recasting these expressions in dimensionless form (and omitting overall multiplying constants), we find alternatively that

\[ \frac{1}{2} \left( \mathbf{r}^2 + \mathbf{y}^2 \right) - \frac{1}{\sqrt{x^2 + y^2}} = \text{constant} \quad ; \quad \mathbf{r} \mathbf{v}_y - \mathbf{y} \mathbf{v}_x = \text{constant} \]

(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
\[
    \frac{d^2 f}{dx^2} + k^2 f = 0 \quad (11.56)
\]
where \( k^2 = \omega^2/c^2 \). If, finally, the string is firmly tied down at two points, say \( x = 0 \) and \( x = \ell \), then this second-order, homogeneous, constant-coefficient, differential equation must be solved subject to the boundary conditions
\[
    f(0) = 0 \quad ; \quad f(\ell) = 0 \quad (11.57)
\]
and we conclude that waves in a string fixed at two points are described by a boundary value problem. Ultimately, we shall be able to find acceptable solutions only for a discrete set of special values of \( k \).

### 11.1.9 The Schrödinger Equation in One Dimension

A quantum mechanical particle of mass \( m \) having definite energy \( E \) and confined in one dimension \( x \) by a potential energy \( V(x) \) is described by a wave function \( \psi(x) \) that satisfies the one-dimensional time-independent Schrödinger equation
\[
    -\frac{\hbar^2}{2m} \frac{d^2 \psi}{dx^2} + V(x) \psi = E \psi \quad \Longrightarrow \quad \frac{d^2 \psi}{dx^2} + \frac{2m}{\hbar^2} \left( E - V(x) \right) \psi = 0 \quad (11.58)
\]
where \( \hbar \) is Planck’s constant divided by \( 2\pi \). We anticipate subsequent exploration of two specific cases. Suppose, for example, that the potential energy is infinite except in the interval \(-\ell \leq x \leq +\ell\), in which interval it is zero, i.e., suppose the particle is confined in an infinitely deep potential well that extends over the specified interval. Then, the wave function describing this particle must satisfy the Schrödinger equation with \( V = 0 \) inside this interval, and it must go to zero at each end of the interval. We seek solutions to the boundary value problem
\[
    \frac{d^2 \psi}{dx^2} + \frac{2mE}{\hbar^2} \psi = 0 \quad ; \quad \psi(0) = \psi(\ell) = 0 \quad (11.59)
\]
Especially if we substitute the shorthand \( k^2 = 2mE/\hbar^2 \) (or \( E = \hbar^2 k^2/2m \)), we recognize that this quantum mechanical problem is mathematically identical to the classical standing wave problem described in Section 11.1.8. This problem admits solutions only for a discrete set of special values for \( k \), i.e., only for special energies.

Suppose, alternatively, that we take the potential energy to be \( V(x) = \frac{1}{2} kx^2 \), which describes the quantum analog to the classical harmonic oscillator with spring constant \( k \). In this case, the time-independent Schrödinger equation becomes
\[
    \frac{d^2 \psi}{dx^2} + \frac{2m}{\hbar^2} \left( E - \frac{1}{2} kx^2 \right) \psi = 0 \quad (11.60)
\]
or, in dimensionless form,
\[
    \frac{d^2 \psi}{dx^2} + (2\epsilon - \bar{x}^2) \psi = 0 \quad (11.61)
\]
where \( \omega = \sqrt{k/m} \), \( \epsilon = E/\hbar \omega \) and \( \bar{x} = x/\sqrt{\hbar/m\omega} \). 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 $x^{10}$. 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.$^{11}$ 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

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

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

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

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

$$\mathcal{L} \{ f(t) \} = \tilde{f}(s) = \int_0^\infty e^{-st} f(t) \, dt \quad (11.64)$$

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

$$m[s^2 \tilde{z}(s) - sz_0 - vz_0] = -\frac{mg}{s} - b[s \tilde{z}(s) - z_0] \quad (11.65)$$

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

$$\tilde{z}(s) = \frac{z_0}{s} + \frac{vz_0s - g}{s^2(s + b/m)} \quad (11.66)$$

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

$$\ddot{z}(s) = \frac{z_0}{s} - \frac{mg/b}{s^2} + \frac{m}{b} \left( v_{z0} + \frac{mg}{b} \right) \left( \frac{1}{s} - \frac{1}{s + b/m} \right)$$

(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

$$z(t) = z_0 - \frac{mg}{b} t + \frac{m}{b} \left( v_{z0} + \frac{mg}{b} \right) \left( 1 - e^{-bt/m} \right)$$

(11.68)

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

11.5 Solving ODEs Symbolically with Mathematica

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 DSolve, which is the primary ODE-solving command in Mathematica, were discussed briefly in Section 8.8.9. In that section, we used DSolve only for solving a single ODE, and we learned that—to that end—the command has two forms, each of which itself has two (sub)forms. To use the first form, we submit either the statement

```
DSolve[ ODE, DVar[IVar], IVar ]
```

or the statement

```
DSolve[ ODE, DVar, IVar ]
```

In either case, the command returns the solution of the single ordinary differential equation ODE for the dependent variable DVar as a function of the single independent variable IVar. In the first subform, the solution is presented as a transformation rule for $x[t]$; in the second subform, Mathematica instead returns a rule defining $x$ as a function of the variable $t$. In the form of this paragraph, DSolve does not impose initial conditions; they haven’t even been specified. Instead, it uses the symbols C[1], C[2], ..., (however many are needed) for the integration constants. These constants must then be determined by subsequent imposition of appropriate initial conditions.

In the second form, the command DSolve accepts as its first argument a list which contains both the differential equation to be solved and an appropriate number of initial conditions. The statement would then have either the first subform

```
DSolve[ { ODE, IC1, IC2, ...}, DVar[IVar], IVar ]
```

or the second subform

```
DSolve[ { ODE, IC1, IC2, ...}, DVar, IVar ]
```

where these statements return results with the same features as described in the previous paragraph for the first form. These forms return solutions on which the initial conditions IC1, IC2, ... have been imposed and which therefore contain 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 $y'[0]==v_0$, where $y_0$ and $v_0$
are the desired values—and may be symbolic or explicitly numeric. Note, in particular, the use of the symbol \( == \) to specify those initial conditions.

The command \texttt{DSolve} is also capable of solving (some) systems of ordinary differential equations. In this case, the first argument will be a \texttt{list} containing all of the differential equations as well as any initial conditions to be imposed, the second argument will be a \texttt{list} containing the several dependent variables, and the third argument will identify the independent variable. In brief, the statement invoking \texttt{DSolve} will have the subform

\[
\text{DSolve}[ \{ ODE1, ODE2, \ldots, IC1, IC2, \ldots \}, \{ DVar1[IVar], DVar2[IVar], \ldots \}, IVar ]
\]

or, alternatively, the subform

\[
\text{DSolve}[ \{ ODE1, ODE2, \ldots, IC1, IC2, \ldots \}, \{ DVar1, DVar2, \ldots \}, IVar ]
\]

### 11.5.1 Projectile in a Viscous Medium

For the sake of a comparison with the result obtained in Section 11.2, we begin by using \textit{Mathematica} to solve the third member of Eq. (11.4). The necessary commands are\footnote{As in Chapter 8, we abbreviate the presentation of \textit{Mathematica} dialogs by making liberal use of terminating semicolons to suppress intermediate output. You are urged to duplicate the dialog in an actual session with \textit{Mathematica}, omitting all terminating semicolons.}

\[
\begin{align*}
\text{In}[1]:= & \quad \text{eq} = m z''[t] == -m g - b z'[t] \\
\text{Out}[1]= & \quad m z''[t] == -m g - b z'[t] \\
\text{In}[2]:= & \quad \text{soln} = \text{DSolve}[ \text{eq}, z, t ][[1,1]] \\
\text{Out}[2]= & \quad z \rightarrow \text{Function}[\{t\}, -\frac{m g t}{b} - \frac{e^{-bt/m} m C[1]}{b} + C[2]]
\end{align*}
\]

Note that \textit{Mathematica} has here tacitly 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

\[
\begin{align*}
\text{In}[3]:= & \quad \text{eq1} = z[0] == z_0 /. \text{soln}; \\
\text{In}[4]:= & \quad \text{eq2} = z'[0] == v z_0 /. \text{soln}; \\
\text{In}[5]:= & \quad \text{Solve}[ \{ \text{eq1,eq2}, \{ C[1], C[2] \} \} ][[1]]; \\
\end{align*}
\]

\[
\begin{align*}
\text{In}[6]:= & \quad \text{soln} = \text{soln} /. \%; \\
\text{In}[7]:= & \quad z[t] /. \text{soln}; \\
\text{In}[8]:= & \quad z'[t] /. \text{soln};
\end{align*}
\]

The end result of these operations is the solution

\[
z(t) = z_0 - \frac{m g}{b} t + \frac{m}{b} \left( v z_0 + \frac{m g}{b} \right) \left( 1 - e^{-bt/m} \right)
\]
for the position and the solution

\[ v_z(t) = -\frac{mg}{b} + \left( v_{z0} + \frac{mg}{b} \right) e^{-bt/m} \]  

(11.70)

for the velocity, though we have rearranged the solutions presented by Mathematica to improve their readability. 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 Mathematica generates. To this end, we substitute the solution into the original equation, evaluate the derivatives explicitly, and simplify the result with the statement

\[
\text{In}[9]:= \text{Simplify}\left[ \text{eq} /. \text{soln} \right] \\
\text{Out}[9]= \text{True}
\]

To complete the verification, we check satisfaction of the initial conditions with the statement

\[
\text{In}[10]:= \text{Simplify}\left[ \{ z[0], z'[0] \} /. \text{soln} \right] \\
\text{Out}[10]= \{ z_0, v_{z0} \}
\]

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

\[
\text{In}[11]:= \text{Collect}\left[ \text{Series}\left[ z[t] /. \text{soln}, \{ b, 0, 1 \} \right], t \right] \\
\text{Out}[11]= z_0 + v_{z0} t - \frac{1}{2} gt^2 - \frac{b t^2}{2m} \left( v_{z0} - \frac{1}{3} gt \right) + O[b]^2
\]

\[
\text{In}[12]:= \text{Collect}\left[ \text{Series}\left[ z'[t] /. \text{soln}, \{ b, 0, 1 \} \right], t \right] \\
\text{Out}[12]= v_{z0} - gt - \frac{b t}{m} \left( v_{z0} - \frac{1}{2} gt \right) + O[b]^2
\]

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 Mathematica 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

\[
\text{In}[13]:= \text{Quit}[] \\
\text{In}[1]:= \text{eq} = m \cdot z''[t] == -m \cdot g - b \cdot z'[t]; \\
\text{In}[2]:= \text{soln} = \text{DSolve}\left[ \{ \text{eq}, z[0]==z_0, z'[0]==v_{z0} \}, z, t \right]; \\
\text{In}[3]:= \text{Quit}[]
\]

These statements lead us in rather fewer steps to the result already presented in Eq. (11.69), though it may require extraction of the expression for the solution from \text{soln} and a subsequent application of Expand and Collect to reveal that agreement more clearly.

---

\[15\] Again, we have taken the liberty to rearrange the terms in Mathematica’s output to facilitate recognition of the limits.
11.5.2 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 Mathematica statements

\begin{verbatim}
In[1]:= eq = n'[t] == k*n[t]*(1-n[t]/nc);

In[2]:= soln = DSolve[ {eq, n[0]==n0}, n, t ]

Out[2]= 
{{n -> Function[{t}, e^kt n0 nc - n0 + e^kt n0 + n_c]}}

In[3]:= soln = soln[[1,1]]

We can verify satisfaction of the initial condition and the differential equation with the statements

Out[4]= n0

Out[5]= True

Verifying the limit as $t \to \infty$ is more complicated because, in the most straightforward statement, Mathematica's command Limit will assume $k$ to be complex and the command has no feature allowing us to tell Mathematica otherwise. Instead, we use the statements

\begin{verbatim}
In[6]:= qn = n[t] /. soln

Out[6]= e^kt n0 nc - n0 + e^kt n0 + n_c

In[7]:= qn /. Exp[k*t] -> tmp

Out[7]= tmp n0 nc - n0 + tmp n0 + n_c

\end{verbatim}

to obtain an expression giving the solution and then cast that expression in terms of the single variable tmp for $e^{kt}$. Then, since $k > 0$ and—hence—$e^{kt} \to \infty$ as $t \to \infty$, we can determine the solution at large $t$ with the statement

\begin{verbatim}
In[8]:= Limit[ %, tmp -> Infinity ]

Out[8]= nc

As expected, the solution approaches the carrying capacity, regardless of the initial population. The solution therefore 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. We have in line Out[6] already obtained the expression to be plotted. We proceed with the statements

\begin{verbatim}
In[9]:= qn = qn/nc

Out[9]= e^kt n0 / -n0 + e^kt n0 + n_c

\end{verbatim}

Measure population $n$ in units of $n_c$. 

Recast to measure *initial* population in units of \(n_c\).

Finally, we make one further recasting to measure time in units of \(1/k\) with the statement

\[
\text{In[11]} := \text{qn = qn} /. \text{t -> } \text{\tau}/k
\]

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

\[
\text{In[12]} := \text{q1 = qn} /. \text{\alpha} \rightarrow 0.25;
\]

\[
\text{In[13]} := \text{q2 = qn} /. \text{\alpha} \rightarrow 0.5;
\]

\[
\text{In[14]} := \text{q3 = qn} /. \text{\alpha} \rightarrow 1.0;
\]

\[
\text{In[15]} := \text{q4 = qn} /. \text{\alpha} \rightarrow 2.0;
\]

\[
\text{In[16]} := \text{q5 = qn} /. \text{\alpha} \rightarrow 4.0;
\]

\[
\text{In[17]} := \text{Plot[ \{q1,q2,q3,q4,q5\}, \{\text{\tau}, 0.0, 5.0\}, PlotRange -> \{0.0, 4.0\}, PlotStyle -> Thickness[0.01], PlotLabel->StyleForm[ "Logistic Behavior", FontFamily->"Times", FontSize -> 14, FontWeight->"Bold"], AxesLabel->{"kt", "n/nc"} ]}
\]

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

\[
\text{In[18]} := \text{q6 = qn} /. \text{\alpha} \rightarrow 0.05;
\]

\[
\text{In[19]} := \text{Plot[ q6, \{\text{\tau}, 0.0, 10.0\}, PlotRange -> \{0.0, 1.0\}, PlotLabel->StyleForm[ "Logistic Behavior", FontFamily->"Times", FontSize -> 14, FontWeight->"Bold"], AxesLabel->{"kt", "n/nc"} ]}
\]

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.

### 11.5.3 Damped Harmonic Oscillator

A third example of the use of *Mathematica* to solve a single second-order, linear, constant-coefficient, homogeneous, ordinary differential equation was presented in Chapter 8. You are invited to review the discussion of the damped harmonic oscillator in Sections 8.8.9 and 11.1.4.

### 11.5.4 Chain Radioactive Decay

As a first example of the use of *Mathematica* 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 *Mathematica* statements
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.

Figure 11.8: Logistic growth when the population is initially much smaller than the carrying capacity. This curve is called the *sigmoid* curve.

\[ \begin{align*}
\text{In[1]} & \text{:= eqA} = A'[t] = -kA*A[t]; \\
\text{In[2]} & \text{:= eqB} = B'[t] = kA*A[t] - kB*B[t]; \\
\text{In[3]} & \text{:= eqC} = C'[t] = kB*B[t]; \\
\text{In[4]} & \text{:= soln} = \text{DSolve[ } \{ \text{eqA, eqB, eqC, } \\
& \text{A[0]==A0, B[0]==0, C[0]==0}, \\
& \text{ } \{ \text{A,B,C}, t \} \} \text{ ]} \\
\end{align*} \]

*Mathematica* displays the fairly complicated result.
To cast these results in a more interpretable form, let us extract each solution as an expression with the statements

In[5]:= \( qA = A[t] \/. \text{soln[[1]]} \)

Out[5]= \( A_0 e^{-k_A t} \)

In[6]:= \( qB = \text{Simplify}[B[t] \/. \text{soln[[1]]}] \)

Out[6]= \( A_0 \left( e^{-k_A t} - e^{-k_B t} \right) / (-k_a + k_B) \)

In[7]:= \( qC = \text{Simplify}[C[t] \/. \text{soln[[1]]}] \)

Out[7]= \( A_0 \left( (k_A - e^{-k_A t}) e^{-k_A t} + (-1 + e^{-k_A t}) k_B \right) / (-k_a + k_B) \)

These forms make evident one important conclusion: 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 graph of \( A(t) \), \( B(t) \), and \( C(t) \) for a specific set of values. Anticipating that we will set \( k_A = k_B \) for the graphs and noting that the expressions obtained for \( B(t) \) and \( C(t) \) are indeterminate when \( k_A = k_B \), we prepare for the graph by evaluating the limits of \( qB \) and \( qC \) as \( k_B \to k_A \). The statements

In[8]:= \( qBeq = \text{Limit}[qB, k_B \to k_A] \)

Out[8]= \( A_0 e^{-k_A t} \)

In[9]:= \( qCeq = \text{Limit}[qC, k_B \to k_A] \)

Out[9]= \( A_0 k_A e^{-k_A t} (-1 + e^{-k_A t}) / (-k_a + k_B) \)

will evaluate those limits, leaving each expressed in terms of the parameters \( k_A \) and \( A_0 \). Then, to obtain the desired graph, we specify values for \( A_0 \) and \( k_A \) with the statements

In[10]:= \( A0 = 1000; kA = 0.1; \)

and produce the graph with the single statement

In[11]:= \( \text{Plot}[\{qA,qBeq,qCeq\}, \{t, 0.0, 50.0\}, \text{PlotRange} \to \{0.0, 1000.0\}, \text{PlotStyle} \to \text{Thickness}[0.005], \text{PlotLabel} \to \text{StyleForm["Radioactive Decay", FontFamily->"Times", FontSize \to 14, FontWeight->"Bold"], \text{AxesLabel} \to \{"t", "A,B,C\}]} \)

The resulting graph is shown in Fig. 11.9.
11.5. SOLVING ODES SYMBOLICALLY WITH MATHEMATICA

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.

11.5.5 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.\(^{16}\) First, suppressing the display of Mathematica’s response, we enter the equations with the statements\(^{17}\)

\[
\text{In}[1]:= \text{eq1 = } x1''[t] + x1[t] - \kappa (x2[t] - x1[t]) == 0; \\
\text{In}[2]:= \text{eq2 = } x2''[t] + x2[t] + \kappa (x2[t] - x1[t]) == 0;
\]

Then we create a list containing the initial values with the statement

\[
\text{In}[3]:= \text{ics = \{ x1[0]==x10, x2[0]==x20, x1'[0]==0, x2'[0]==0 \};}
\]

and, finally, we request the solution with the statement

\[
\text{In}[4]:= \text{soln = DSolve[ Union\{eq1,eq2\}, ics\,\{x1, x2\}, t \]}
\]

While correct, the (suppressed) output from this statement is unfortunately in a form that is difficult to fathom. To rectify that situation, we

- execute the statement

\[
\text{In}[5]:= \text{soln1 = soln /. Sqrt[-1-2*\kappa] -> I*Sqrt[1+2*\kappa]};
\]

\(^{16}\)You are urged to explore other initial conditions.\(^{17}\)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.
to replace several occurrences of $\sqrt{-1 - 2\kappa}$ with $i\sqrt{1 + 2\kappa}$, thus making the imaginary character of $\sqrt{-1 - 2\kappa}$ explicit. (Remember that $\kappa > 0$.)

- extract expressions for the position of each oscillator as a function of time with the statements

  ![Mathematica Code](image)

- execute the statements

  ![Mathematica Code](image)

- convert complex exponentials to trigonometric functions with the statements

  ![Mathematica Code](image)

At this point, we recognize that the time dependence of each term in these solutions has either $\cos t$ or $\cos [t\sqrt{1 + 2\kappa}]$ as a factor. Thus, we collect those terms with the nested statements

![Mathematica Code](image)

At this point, we finally recognize that 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).

### 11.5.6 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 enter and solve the equation subject to zero boundary conditions at each end of the string with the statements

![Mathematica Code](image)
11.5. SOLVING ODES SYMBOLICALLY WITH MATHEMATICA

The version of *Mathematica* here in use recognizes that only the trivial solution exists unless $k$ has one of the special values $k = n\pi/l, n \in \text{integer}$. In the special case, we learn that

$$f(x) = C_1 \sin \frac{n\pi x}{l}$$

(11.71)

where $C_1$ is undetermined and $n$ 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 simply as an alternative choice of $C_1$). Further, since $\omega = kc$, we would associate with this solution the frequency

$$\omega_n = kc = \frac{\pi c}{l} = n \omega_1$$

(11.72)

where $\omega_1$ is the fundamental frequency. We find, in particular, that all allowed frequencies are integer multiples of the fundamental frequency.

Earlier versions of *Mathematica* concluded that only the trivial solution to the ODE can be made to satisfy both boundary conditions and returned only the solution

Out[2]= \{ \{ \text{f} \rightarrow \text{Function}[(x), 0] \} \}

to the statement at In[2] above. These versions of *Mathematica* failed to recognize that we can find non-trivial solutions, but only if we are prepared to constrain $k$ to a limited set of values.

In those versions, we must evidently be more deliberate. To help *Mathematica* 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. First, we solve the equation with the statement

In[3]\[ soln = \text{DSolve}[\text{eq}, \text{f}, x][[1,1]] \]


raising the solution with \([1,1]\) to the top level in its expression. Next, we determine the consequences of the boundary conditions with the statements

In[4]\[ \text{eq1} = 0 == \text{f}[0] /. \text{soln} \]

\text{Impose condition at } x = 0.

Out[4]= 0 == C[1]

In[5]\[ \text{eq2} = 0 == \text{f}[l] /. \text{soln} \]

\text{Impose condition at } x = l.


At this point, we recognize that we have a pair of equations to be solved simultaneously for $C[1]$ and $C[2]$. 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 for $C[1]$ and $C[2]$. To find the coefficient matrix, we invoke the *Mathematica* statement\(^{18}\)

In[5]\[ \text{tmp} = \text{CoefficientArrays}[\{\text{eq1, eq2}\}, \{C[1], C[2]\}] \]

Out[5]= \{ \text{SparseArray[ } mess \text{ ], SparseArray[ } mess \text{ ]} \}

whose output is a bit obscure. Even so, the statement\(^{19}\)

\(^{18}\)In some earlier versions of *Mathematica*, the necessary functionality resided in the older command \texttt{LinearEquationsToMatrices} in the package \texttt{LinearAlgebra}. While still available, that entire package is now obsolete and may not be included in future versions of *Mathematica*.

\(^{19}\)In the output from *Mathematica*, all braces will be the same size. We have here used different sizes to clarify the structure of that output.
yields a more transparent result. Here, the output is a list of lists, the first of which conveys the
inhomogeneities in the equations while the second, itself a list of lists, provides the coefficient matrix
(though the minus signs in several places are a bit mysterious). We can extract and display the
coefficient matrix with the statements

\[
\text{In}[7]:= \text{coefmat} = \text{tmp1[[2]]};
\]
\[
\text{In}[8]:= \text{MatrixForm}[\text{coefmat}]
\]

Next, we seek values of \(k\) for which the determinant of this matrix will be zero by invoking the
statements

\[
\text{In}[9]:= \text{charpol} = \text{Det[coefmat]} \quad \text{Evaluate determinant.}
\]
\[
\text{Out}[9]= \sin[\kappa \ell]
\]
\[
\text{In}[10]:= \text{rts} = \text{Solve[charpol==0, \kappa]} \quad \text{Solve characteristic equation for \kappa.}
\]

\[
\text{Out}[10]= \left\{\begin{array}{l}
\kappa \to \text{ConditionalExpression}\left[\frac{2\pi C[1]}{\ell}, C[1] \in \text{Integers}\right], \\
\kappa \to \text{ConditionalExpression}\left[\frac{\pi + 2\pi C[1]}{\ell}, C[1] \in \text{Integers}\right]
\end{array}\right\}
\]

*Mathematica* recognizes that this equation has numerous roots.\(^{20}\) Note that the \(C[1]\) in these
statements is *not* the same as the \(C[1]\) at Out[2] and In[4]–Out[5] and in Eq. (11.71). To avoid
confusion, let’s simplify the appearance here by replacing *this* \(C[1]\) with \(n\) with the statement

\[
\text{In}[11]:= \text{rts1} = \text{rts} /. C[1] \to n
\]

\[
\text{Out}[11]= \left\{\begin{array}{l}
\kappa \to \text{ConditionalExpression}\left[\frac{2\pi n}{\ell}, n \in \text{Integers}\right], \\
\kappa \to \text{ConditionalExpression}\left[\frac{\pi + 2\pi n}{\ell}, n \in \text{Integers}\right]
\end{array}\right\}
\]

In this form, it is easier to see that, with \(n\) an integer, \(k = 2n\pi/\ell\) or \(k = (2n+1)\pi/\ell\), i.e., \(k\) is either
an even or an odd multiple of \(\pi/\ell\). More simply, \(k = n\pi/\ell\) for any integer. With this recasting,
we use the statement

\[
\text{In}[12]:= \text{rl} = k \to n*\text{Pi}/\ell;
\]

to define a transformation rule for substituting \(n\pi/\ell\) for \(k\) in any expression. With this rule, we
quickly find the corresponding values of \(C[1]\) and \(C[2]\) by returning to equations eq1 and eq2 and
executing the statements

\[
\text{In}[13]:= \text{eq1} /. \text{rl}
\]
\[
\text{Out}[13]= 0 == C[1]
\]
\[
\text{In}[14]:= \text{Simplify[eq2 /. rl, \text{Element}[n,\text{Integers}]]}
\]
\[
\text{Out}[14]= 0 == C[1]
\]

\(^{20}\)An earlier version of *Mathematica* returned the single (correct) root \(k = 0\). The behavior has changed in newer
versions of *Mathematica*. 
Clearly, we must choose \( C[1] \) to be 0—a fact we really knew at the very beginning, but we learn nothing about \( C[2] \), 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 \( n\pi/\ell \) and \( C[1] \) is set to zero. Within Mathematica, we then find that the solution to the original boundary value problem is given by

\[
\text{In}[15]:= \text{stdwv} = f[x] /. \text{soln} \\
\]

\[
\text{In}[15]:= \text{stdwv} = \text{stdwv} /. \{r1, C[1]->0\} \\
\text{Out}[15]= C[2] \sin\left[\frac{n\pi x}{\ell}\right]
\]

where \( n \) assumes any of the values 1, 2, 3, \ldots (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 simply as an alternative choice of \( C[2] \)). After this more arduous route, we arrive again at the solution expressed in Eqs. (11.71) and (11.72).

### 11.5.7 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

\[
E_n = \frac{\hbar^2 k_n^2}{2m} = \frac{n^2 \hbar^2 \pi^2}{2m\ell^2}
\]

(11.73)

Further, we would associate the (unnormalized) wave function

\[
\psi_n(x) = N_n \sin\left[\frac{n\pi x}{\ell}\right]
\]

(11.74)

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.

### 11.5.8 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

\[
f(x) = \sum_{i=0}^{\infty} a_i x^i + \gamma
\]

(11.75)

and seeking both an indicial equation determining \( \gamma \) and recursion relationships determining \( a_i \). In the absence of a simple Mathematica command to facilitate this approach to solving an ODE, we must either write our own package (which is beyond the scope of this book) or step Mathematica systematically through the process. For the sake of concreteness, suppose we seek a solution to the equation

\[
\frac{d^2 y}{dx^2} + y = 0
\]

We might then define a (truncated) series for the solution with the statement

\[
\text{In}[1]:= y[x_] := \text{Sum}[ a[i] x^i \{r1, C[1]->0\}, \{i, 0, 8\} ]
\]
(We could, of course, extend the series as far as we wished. We here truncate it with a few terms to keep the output to manageable size.) Then we would evaluate the right-hand side of the ODE with the statement

\[
\text{In}[2]:= y''[x] + y[x]
\]
\[
\text{Out}[2]= (\text{expression too long to record})
\]

Not surprisingly, each term has a factor of \(x^\gamma\). Since we ultimately will set this expression equal to zero, we can divide the expression by \(x^\gamma\), and then—since a premultiplying factor of \(1/x^2\) appears—multiply the entire expression by \(x^2\). The necessary \textit{Mathematica} statement and its output are

\[
\text{In}[3]:= \text{soln1} = x^2*\text{Simplify}[\ %/x^\{\Gamma\} \]
\]
\[
\text{Out}[3]= (-1 + \gamma)\gamma a_0 + x\gamma(1 + \gamma)a_1 + x^2(a_0 + (2 + 3\gamma + \gamma^2)a_2 + \ldots)
\]

We have written out only the terms with the lowest powers of \(x\) for the moment because they are the terms on which we must focus first. Since this series must be identically zero for all \(x\) and since different powers of \(x\) are linearly independent, the series will sum to zero only if the coefficient of each power of \(x\) is separately zero. To simplify the application of this criterion, we might extract the coefficients themselves as a list with the statement

\[
\text{In}[4]:= \text{coefs} = \text{CoefficientList}[\ \text{soln1}, x]
\]
\[
\text{Out}[4]= \{(-1 + \gamma)\gamma a_0, \gamma(1 + \gamma)a_1, a_0 + (2 + 3\gamma + \gamma^2)a_2, \ldots\}
\]

Following the method of Frobenius, we determine acceptable values of \(\gamma\) by requiring that the coefficient of the lowest power of \(x\) (i.e., the first element in this list) be zero under the restriction that \(a_0 \neq 0\). The statements

\[
\text{In}[5]:= \text{coefs}[[1]]/a[0]
\]
\[
\text{Out}[5]= (-1 + \gamma)\gamma
\]
\[
\text{In}[6]:= \text{gam} = \text{Solve}[\ % == 0, \{\Gamma\}]
\]
\[
\text{Out}[6]= \{\{\gamma \rightarrow 0\}, \{\gamma \rightarrow 1\}\}
\]

reveal that \(\gamma\) can be either 0 or 1. When \(\gamma = 1\), the coefficients of successive powers of \(x\) are

\[
\text{In}[7]:= \text{coefs1} = \text{coefs} /. \text{gam}[[2]]
\]
\[
\text{Out}[7]= \{0, 2a_1, a_0 + 6a_2, a_1 + 12a_3, a_2 + 20a_4, a_3 + 30a_5, a_4 + 42a_6, a_5 + 56a_7, a_6 + 72a_8, a_7, a_8\}
\]

Each of these coefficients must itself be zero. Ignoring the first coefficient (which is already zero) and the last two coefficients (which are incomplete because of our truncation of the series), we extract a set of eight simultaneous linear equations and solve them for the coefficients with the statements

\[
\text{In}[8]:= \text{eqs1} = \text{Table}[\ \text{coefs1}[[i+1]]==0, \{i, 8\}]
\]
\[
\text{Out}[8]= \{2a_1 == 0, a_0 + 6a_2 == 0, a_1 + 12a_3 == 0, \ldots\}
\]
\[
\text{In}[9]:= \text{vars1} = \text{Table}[\ a[i], \{i, 8\}]
\]
\[
\text{Out}[9]= \{a_1, a_2, a_3, \ldots\}
\]

\(^{21}\)To shorten the expression, we use the symbol \(a_i\) rather than \(a[i]\) for the \(i\)-th coefficient in the original series.
11.5. SOLVING ODES SYMBOLICALLY WITH MATHEMATICA

In[10]:= Solve[ eqs1, vars1 ][[1]]
Out[10]= 

Finally, we can substitute these solutions into the original series with the statement

In[11]:= y[x] /. \[Gamma] -> 1;
In[12]:= % /. %10;
In[13]:= Collect[ %, a[0] ]
Out[13]=

Recognizing that 3! = 6, 5! = 120, 7! = 5040, and 9! = 362880, we then also recognize that the series in parentheses is in fact the beginnings of the series for Sin [x] and this solution apparently is a_0 Sin [x].

The solution when \(\gamma = 0\) can be found in much the same way. We begin by invoking the statement

In[14]:= coefs0 = coefs /. gam[[1]]
Out[14]=

to determine the coefficients for this case. As before, each of these coefficients must itself be zero. Ignoring the first two coefficients (which are already zero) and the last two (which are incomplete because of our truncation of the series), we extract a set of seven simultaneous linear equations and solve them for the coefficients with the statements

In[15]:= eqs0 = Table[ coefs0[[i+2]]==0, {i, 7} ]
Out[15]=

In[16]:= vars0 = Table[ a[i+1], {i, 7} ]
Out[16]=

In[17]:= Solve[ eqs0, vars0 ][[1]]
Out[17]=

Here, we have noted that the conditions imposed on the solution do not constrain either a_0 or a_1. Finally, we can substitute these solutions into the original series with the statement

In[18]:= y[x] /. \[Gamma] -> 0;
In[19]:= %/. %17;
In[20]:= Collect[ %, { a[0], a[1] } ]
Out[13]=

This value of \(\gamma\) has, in fact, yielded a solution containing two arbitrary constants, exactly what we expect in the general solution to a second-order ODE. Interestingly, the series multiplied by a_1 in this result is identical to the (only) series obtained when \(\gamma\) was set to the value 1, i.e., it is again the function Sin [x]. Recognizing that 2! = 2, 4! = 24, 6! = 720, and 8! = 40320, we recognize that the other series in this result—the one multiplied by a_0—is in fact the beginnings of the series for Cos [x]. Thus, the solution obtained by the method of Frobenius is apparently a_0 Cos [x] + a_1 Sin [x], where a_0 and a_1 are to be determined by initial or boundary conditions.
11.6 Algorithms for Solving ODEs Numerically

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

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

\[ \frac{dx}{dt} = f(x, t) \quad ; \quad x(0) = x_0 \]  \hspace{1cm} (11.76)

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, we can write the approximation

\[ \frac{dx}{dt} = f(x, t) \quad \Rightarrow \quad \frac{x(t + \Delta t) - x(t)}{\Delta t} \approx f(x, t) \quad \Rightarrow \quad x(t + \Delta t) \approx x(t) + f(x, t) \Delta t \]  \hspace{1cm} (11.77)

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

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

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.\(^{23}\) As this example shows, we can generate a complete—though approximate—solution from initial knowledge of only

1. The differential equation, which determines the rate of change of the dependent variable from the dependent and independent variables themselves,

2. The initial condition, which starts the process by providing the first row in a table containing \( t, x, \) and \( f \).

\(^{22}\) 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.

\(^{23}\) 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.
11.6. ALGORITHMS FOR SOLVING ODES NUMERICALLY

Table 11.1: Simple Euler algorithm.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Expression</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>dt</td>
<td>( \Delta t )</td>
<td>Choose time step.</td>
</tr>
<tr>
<td>told</td>
<td>0.0</td>
<td>Initialize ( t_{old} ).</td>
</tr>
<tr>
<td>xold</td>
<td>( x_0 )</td>
<td>Initialize ( x_{old} ).</td>
</tr>
<tr>
<td>loop</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fold</td>
<td>( f(x_{old}, t_{old}) )</td>
<td>Evaluate ( f_{old} ).</td>
</tr>
<tr>
<td>exit_loop when done</td>
<td></td>
<td>Display results.</td>
</tr>
<tr>
<td>xnew</td>
<td>( x_{old} + fold \cdot dt )</td>
<td>Calculate values at new time.</td>
</tr>
<tr>
<td>tnew</td>
<td>( t_{old} + dt )</td>
<td>Replace old values with new.</td>
</tr>
<tr>
<td>xold</td>
<td>( x_{new} )</td>
<td></td>
</tr>
<tr>
<td>told</td>
<td>( t_{new} )</td>
<td></td>
</tr>
<tr>
<td>end_loop</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

\[
A(t + \Delta t) = A(t) + \frac{dA}{dt}(t) \Delta t = A(t) - k_A A(t) \Delta t \tag{11.79}
\]

\[
B(t + \Delta t) = B(t) + \left[ k_A A(t) - k_B B(t) \right] \Delta t \tag{11.80}
\]

\[
C(t + \Delta t) = C(t) + k_B B(t) \Delta t \tag{11.81}
\]

For definiteness,\(^{24}\) 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 \( \Delta t = 0.250 \).\(^{25}\) Equations (11.79), (11.80), and (11.81) at \( t = 0.0 \) then become

\[
A(0.00 + 0.25) = 1000.0 - 0.1 \cdot 1000.0 \cdot 0.25 = 975.000 = A(0.25)
\]

\[
B(0.00 + 0.25) = 0.0 + \left[ 0.1 \cdot 1000.0 - 0.1 \cdot 0.0 \right] \cdot 0.25 = 25.000 = B(0.25)
\]

\[
C(0.00 + 0.25) = 0.0 + 0.1 \cdot 0.0 \cdot 0.25 = 0.000 = C(0.25)
\]

Continuing the algorithm further by applying Eqs. (11.79), (11.80), and (11.81) for \( t = 0.25 \), we find that

\[
A(0.25 + 0.25) = 975.0 - 0.1 \cdot 975.0 \cdot 0.25 = 950.625 = A(0.50)
\]

\[
B(0.25 + 0.25) = 25.0 + \left[ 0.1 \cdot 975.0 - 0.1 \cdot 25.0 \right] \cdot 0.25 = 48.750 = B(0.50)
\]

\[
C(0.25 + 0.25) = 0.0 + 0.1 \cdot 25.0 \cdot 0.25 = 0.625 = C(0.50)
\]

\(^{24}\)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.

\(^{25}\)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>1000.00</td>
<td>0.000</td>
<td>0.000</td>
<td>1000.000</td>
</tr>
<tr>
<td>0.25</td>
<td>975.000</td>
<td>25.000</td>
<td>0.000</td>
<td>1000.000</td>
</tr>
<tr>
<td>0.50</td>
<td>950.625</td>
<td>48.750</td>
<td>0.625</td>
<td>1000.000</td>
</tr>
<tr>
<td>0.75</td>
<td>926.859</td>
<td>71.297</td>
<td>1.844</td>
<td>1000.000</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Going yet one more step, Eqs. (11.79), (11.80), and (11.81) for $t = 0.50$ yield that

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

The resulting values are compiled in Table 11.2, in which the first row is provided by the initial conditions and all subsequent rows are determined by the algorithm outlined in Eqs. (11.79), (11.80), and (11.81). 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.\footnote{Actually, however, that the sum $A + B + C$ preserves its initial value is not really a check on the accuracy of the method in this example. If we simply add the three stepping equations we are using [Eqs. (11.79)–(11.81)], we find that $A(t + \Delta t) + B(t + \Delta t) + C(t + \Delta t) = A(t) + B(t) + C(t)$. Solutions generated by Euler’s method applied to this problem automatically satisfy the conservation law, regardless of $\Delta t$.}

### 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 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
   \[
   x_{\text{pred}} = x_{\text{old}} + f_{\text{old}} \Delta t ; \quad t_{\text{new}} = t_{\text{old}} + \Delta t
   \]  
3. calculate $f_{\text{pred}} = f(x_{\text{pred}}, t_{\text{new}})$, and
4. calculate the final (corrected) value
   \[
   x_{\text{new}} = x_{\text{old}} + \frac{1}{2} \left( f_{\text{old}} + f_{\text{pred}} \right) \Delta t
   \]
Table 11.3: Improved Euler algorithm.

\[
\begin{align*}
\text{dt} & \leftarrow \Delta t \\
\text{told} & \leftarrow 0.0 \\
\text{xold} & \leftarrow x_0 \\
\text{loop} & \\
\text{fold} & \leftarrow f(\text{xold}, \text{told}) \\
\text{print}, \text{told}, \text{xold}, \text{fold} & \\
\text{exit} & \text{loop when done} \\
\text{xpred} & \leftarrow \text{xold} + \text{fold} \times \text{dt} \\
\text{tnew} & \leftarrow \text{told} + \text{dt} \\
\text{fpred} & \leftarrow f(\text{xpred}, \text{tnew}) \\
\text{xnew} & \leftarrow \text{xold} + \frac{1}{2}(\text{fold} + \text{fpred}) \times \text{dt} \\
\text{xold} & \leftarrow \text{xnew} \\
\text{told} & \leftarrow \text{tnew} \\
\text{end} & \text{loop}
\end{align*}
\]

Choose time step. 
Initialize \( t_{\text{old}} \). 
Initialize \( x_{\text{old}} \). 
Evaluate \( f_{\text{old}} \). 
Display results. 
Calculate predicted value. 
Evaluate \( f_{\text{pred}} \). 
Calculate final (corrected) value. 
Replace old values with new.

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>
<tr>
<td>\ldots</td>
<td>\ldots</td>
<td>\ldots</td>
<td>\ldots</td>
<td>\ldots</td>
</tr>
</tbody>
</table>

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 \( f_{\text{old}} \) at the beginning of the interval and (b) the best estimate \( f_{\text{pred}} \) 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.\(^{27}\)

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.

\(^{27}\)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.
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

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

\[
= x(t) + f(x,t) \Delta t + \frac{1}{2} \frac{df(x,t)}{dt} \Delta t^2 + O(\Delta t^3)
\]

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

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

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. (11.87). To deduce this second Taylor series, we first invoke the two-dimensional Taylor series to find that

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

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

(11.88)

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

\[
x(t + \Delta t) = x(t) + w_1 f(x,t) \Delta t
\]

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

\[
= x(t) + \left( w_1 + w_2 \right) f(x,t) \Delta t
\]

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

(11.89)

which will match Eq. (11.87) through terms of order \( O(\Delta t^2) \) if we choose

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

(11.90)

\[28\]Note that the steps laid out here reduce to the improved Euler method if we take \( a = b = 1 \) and \( w_1 = w_2 = 1/2 \).
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 28—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

$$t' = t + \frac{1}{2} \Delta t; \quad x' = x + \frac{1}{2} f(x, t) \Delta t; \quad x(t + \Delta t) = x(t) + f(x', t') \Delta t$$

(11.91)

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:

\begin{align*}
&k_1 \leftarrow f(x_{\text{old}}, t_{\text{old}}) \Delta t \quad \text{Find change using rate at beginning.} \\
&k_2 \leftarrow f((x_{\text{old}} + k_1) / 2, (t_{\text{old}} + \Delta t / 2) \Delta t) \quad \text{Estimate change using rate at midpoint.} \\
&k_3 \leftarrow f((x_{\text{old}} + k_2) / 2, (t_{\text{old}} + \Delta t / 2) \Delta t) \quad \text{Estimate change using refined rate at midpoint.} \\
&k_4 \leftarrow f(x_{\text{old}} + k_3, t_{\text{old}} + \Delta t) \Delta t \quad \text{Estimate change using rate at end.} \\
&x_{\text{new}} \leftarrow x_{\text{old}} + (k_1 + 2k_2 + 2k_3 + k_4) / 6 \quad \text{Calculate final value.}
\end{align*}

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

---

29Actually, we choose only $a$. Then $b = a$, $w_2 = 1/2a$, and $w_1 = 1 - w_2$ are fixed.
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 $10^5$. Indeed, the second value is probably better than that, but assessing _its_ accuracy by this
method would entail obtaining a _third_ value by using a 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

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

(11.92)

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

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

(11.93)

or that

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

(11.94)

30This assertion is refined in the next paragraphs.
31Warning: This approach is not entirely secure if the procedure converges slowly as the time step is reduced.
Remember that $\sum_{n=1}^{\infty} 1/n$ diverges even though the effect of the millionth term added to the partial sum starts in
the sixth decimal place.
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. (11.93) can be deduced for all of the methods that we have described in this section. Without proof, we present the properties

\[ x_{\text{exact}}(t + \Delta t) = x_{\text{Euler}}(t + \Delta t) + O(\Delta t^2) \] (11.95)
\[ = x_{\text{ImpEul}}(t + \Delta t) + O(\Delta t^3) \] (11.96)
\[ = x_{\text{RK2}}(t + \Delta t) + O(\Delta t^3) \] (11.97)
\[ = x_{\text{RK4}}(t + \Delta t) + O(\Delta t^5) \] (11.98)

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

Unfortunately, the analysis described briefly in the previous paragraph is not the whole story. Each of the statements in that paragraph is correct, provided we assume that the step that arrives at the various estimates of \( x(t + \Delta t) \) starts with the exact solution at time \( t \). Except for the first step, which moves away from (exact) initial conditions, that assumption is invalid. Truncation errors per step compound as more and more steps are taken, and a full assessment of the error in a solution obtained by one or another numerical means must recognize this cumulation. Beyond the error per step, we must be aware of the global truncation error, which attempts to estimate how much error accumulates in the course of working out a solution over the entire desired range of the independent variable. The task of assessing global error is extremely difficult. Crudely, however, if we simply add up expressions like those in the previous paragraph for the \( N \) steps in an entire solution and suppose—without much justification—that each step contributes about the same amount, we would conclude that the global truncation error would be order \( NO(\Delta t^p) \) 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, four, and sixteen, respectively. In particular, halving the step size with the fourth-order Runge-Kutta method will add at least one more decimal digit to the accuracy of the solution overall (and may well do much better).

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

### 11.6.5 Adaptive Methods

In the previous subsections, we assumed that the user of a particular algorithm would actually view the values obtained for different time steps and decide personally when to stop by examining the changes that occur as the time step is successively halved. One can, of course, program a computer to make those decisions. In essence, the program generates the solution with one time step, then

\[^{33}\text{M. Abramowitz and I. A. Stegun, Handbook of Mathematical Functions, (United States) National Bureau of Standards Applied Mathematics Series, No. 55, June, 1964, pg. 896–897.}\]
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. (11.76), 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

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

(11.99)

(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. (11.99), we might use a fourth-order Runge-Kutta method to obtain $x_1$, $x_2$, and $x_3$ from $x_0$. Then we could shift to the fourth-order Adams-Bashforth formula for all subsequent points.

34Sometimes 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.

35In 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.

We could, of course, simply take the result given by Eq. (11.99) 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

\[
x_{i+1} = x_i + \frac{\Delta t}{24} \left( 9f_{i+1} + 19f_i - 5f_{i-1} + f_{i-2} \right)
\]  

(11.100)

Once a sufficient number of values is in hand, we could at each step invoke Eq. (11.99) to predict the next solution and then Eq. (11.100) to refine it. We might call the resulting method the Adams-Bashforth-Moulton method.

11.10 Solving ODEs Numerically with PYTHON

*Note:* All PYTHON program (.py) files referred to in this chapter are available in the directory `$HEAD/python`, where (as defined in the Local Guide) `$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 PYTHON’s default search path. If so, the files can be found by PYTHON 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 PYTHON’s’s elementary commands or, more simply, by invoking one of PYTHON’s built-in routines `scipy.integrate.odeint` or `scipy.integrate.ode`. The module `scipy` must, of course, be installed for these features to be invoked.

11.10.1 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 0.0 \( \leq t \leq 50.0 \) with a time step of \( \Delta t = 0.25 \) and \( k_A = k_B = 0.1 \). We could exploit the capabilities of PYTHON as follows. We anticipate that values will be stored in arrays \( t, A, B, \) and \( C \). Ultimately these arrays will have 201 elements (time interval divided by time step = 50.0/0.25 = 200 steps, but we need an element also for the initial value). Initially, however, we set only the first element of each array and, of course, the parameters and the time step. Possible statements are

\[
\text{import numpy as np} \\
t=[0.0]; A=[1000.0] \\
B=[0.0]; C=[0.0] \\
kA = 0.1; kB = 0.1 \\
dt = 0.25
\]

Import needed module.
Set initial values.
Set parameters.
Set time step.

The solution is then calculated, time instant by time instant, with the loop
for i in np.arange(1,201,1):
    B = np.append( B, [B[i-1] + kA*A[i-1]*dt - kB*B[i-1]*dt] )
    C = np.append( C, [C[i-1] + kB*B[i-1]*dt] )
    t = np.append( t, [t[i-1] + dt] )

which calculates each new \(A\), \(B\), and \(C\) using the stepping equations given in Eqs. (11.79), (11.80), and (11.81) and each new \(t\) simply by adding the time step \(dt\) to the previous \(t\).\(^{37}\) The simple statement

for i in range(4):
    print( '{0:10.2f}{1:10.3f}{2:10.3f}{3:10.3f}{4:10.3f}'.
        format(t[i],A[i],B[i],C[i],A[i]+B[i]+C[i]) )

0.00 1000.000 0.000 0.000 1000.000
0.25 975.000 25.000 0.000 1000.000
0.50 950.625 48.750 0.625 1000.000
0.75 926.859 71.297 1.844 1000.000

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

import matplotlib.pyplot as plt
plt.plot( t, A, color='black', linewidth=3 )
plt.plot( t, B, color='black', linewidth=3 )
plt.plot( t, C, color='black', linewidth=3 )
plt.title( 'Three-Species Decay Chain', fontsize=20 )
plt.xlabel( 'Time', fontsize=14 )
plt.ylabel( 'Number of Atoms Present', fontsize=14 )
plt.text( 2.5, 850.0, '$A$', fontsize=14 )
plt.text( 2.5, 300.0, '$B$', fontsize=16 )
plt.text(40.0, 800.0, '$C$', fontsize=16 )
plt.show()

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 two 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 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

\(^{37}\)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 \(t[i] = (i-1) \cdot dt\), 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 11.6: Solution to radioactive decay at $t = 6.0$ for the 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>6.000</td>
<td>2.0000</td>
<td>512.00</td>
<td>384.00</td>
<td>104.00</td>
</tr>
<tr>
<td>1.000</td>
<td>0.0000</td>
<td>531.44</td>
<td>354.29</td>
<td>114.26</td>
</tr>
<tr>
<td>0.500</td>
<td>0.2500</td>
<td>540.36</td>
<td>341.28</td>
<td>118.36</td>
</tr>
<tr>
<td>0.250</td>
<td>0.5000</td>
<td>544.64</td>
<td>335.16</td>
<td>120.19</td>
</tr>
<tr>
<td>0.125</td>
<td>1.0000</td>
<td>546.74</td>
<td>332.20</td>
<td>121.06</td>
</tr>
<tr>
<td>0.0625</td>
<td>2.0000</td>
<td>547.78</td>
<td>330.73</td>
<td>121.49</td>
</tr>
</tbody>
</table>

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.

11.10.2 The PYTHON Command odeint

The simpler of PYTHON’s commands for solving ODEs is scipy.integrate.odeint,\textsuperscript{38} which is patterned after the lsoda component in a large FORTRAN solver (the Livermore Solver for Ordinary

\textsuperscript{38}The commands scipy.integrate.ode and scipy.integrate.solve_ivp are a bit more versatile. See Section 11.10.11 for details on these alternate integrators.
Figure 11.11: Solution of chain radioactive decay via Euler’s method. The solid, dotted, and dashed lines show the solution for $\Delta t = 0.0625, 0.5, \text{ and } 2.0$, 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.

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>549.404</td>
<td>327.821</td>
<td>122.776</td>
<td></td>
</tr>
<tr>
<td>0.5000</td>
<td>548.954</td>
<td>328.940</td>
<td>122.106</td>
<td></td>
</tr>
<tr>
<td>0.2500</td>
<td>548.847</td>
<td>329.202</td>
<td>121.951</td>
<td></td>
</tr>
<tr>
<td>0.1250</td>
<td>548.820</td>
<td>329.266</td>
<td>121.904</td>
<td></td>
</tr>
<tr>
<td>0.0625</td>
<td>548.814</td>
<td>329.282</td>
<td>121.904</td>
<td></td>
</tr>
</tbody>
</table>

Differential Equations), written by Alan Hindmarsh and first appearing in 1980. The simplest statement invoking odeint has the general form:

```python
soln = odeint( FunctionName, InitialConds, IndepVar )
```

where

- `soln`—any legal variable name will do—is the name of the (scalar or list) variable in which the solution is returned.
- `FunctionName` is the name of the function defining the equations to be solved; see Section 11.10.3 for the structure of this definition.

\[39\] We assume that the statement from scipy.integrate import odeint has previously been executed.
• *InitialConds* is the name of the (scalar or list) variable that provides the initial value of each dependent variable. Alternatively, the scalar or list value itself may be provided as this argument to *odeint*.

• *IndepVar* is the name of the list variable that provides the values of the independent variable at which solutions are to be calculated. Alternatively, the list itself may be provided as this argument to *odeint*.

The procedure is adaptive, which means that, in generating the solution, *odeint* varies the step size dynamically as it discovers how rapidly the solution changes in various regions within the full range of the independent variable. As output, the procedure generates solutions at the user-specified values—usually equally spaced—in *IndVar*, but, behind the scenes, *odeint* will be taking steps of various sizes depending on how rapidly the solution changes and the tolerances—default or otherwise—specified when *odeint* is invoked. Because the varying step sizes may not generate solutions at the user-specified values of the independent variable, *odeint* will interpolate when necessary to yield solutions at the requested values of the independent variable.

To speed the actual illustration of the use of *odeint*, we postpone discussing two remaining issues. Sometimes, we will be interested in the dependence of the solution on one or more parameters embedded in the differential equations. Conveniently, *odeint* admits the keyword *args* that provides for setting parameters that *odeint*, when called, will pass through to the function that it calls to determine derivatives. This feature will be introduced towards the end of Section 11.10.4.

Second, the detailed behavior of *odeint*—tolerances imposed, maximum and minimum step sizes, . . .—is controlled by a number of options, each of which has a (judiciously chosen) default value but each of which can also be changed should change be warranted in a particular circumstance. For a while, we will simply accept all default values. Discussion of the value of those defaults and the ways to change them is postponed to Section 11.10.9.  

### 11.10.3 Defining ODEs for *odeint*

Before using the routines PYTHON supplies for solving a system of first-order differential equations numerically, we must create a py-file defining a function whose execution by PYTHON 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 py-file. In broad outline, this file will have the form

```python
def FunctionName( DepVars, IndVar ):
    ...
    Intermediate statements.
    ...
    return [ DerivOfFirstVar, DerivOfSecondVar, . . . ]
```

In this structure,

- The first executable line in the function definition must begin with the keyword *def*.
- The variables must be given in the order shown.
- *DepVars* is a one dimensional array or list that, on input, supplies the values of the dependent variables for which the derivatives are to be computed.

---

40 The URL docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.odeint.html links to a very detailed discussion of *odeint*.

41 See Section 5.6.1.

42 Remember that comments can be introduced either with # or %. See item 6 in Section 5.1.
• \textit{IndVar} is a scalar that, on input, supplies the value of the independent variable at which the derivatives are to be computed.
• Don’t fail to notice the colon at the end of the first line.
• All lines within the body of the definition must be properly indented.
• For convenience, we shall store this function definition in a file having the same name as we assign to the function. For example, if we choose to call the function for the radioactive decay chain, \texttt{decay}, then the file storing the function will be named \texttt{decay.py}.

Explicit .py-files for several examples will be presented in the subsequent subsections.

11.10.4 Chain Radioactive Decay

As a first example of the use of \texttt{odeint}, consider the equations describing chain radioactive decay, Eq. (11.19). Before we can construct the proper .py-file to communicate the differential equations to PYTHON’s routine, we must establish a correspondence between the elements of a list—name it \texttt{n}—and the dependent variables. Since we have three dependent variables (the number of atoms of species \(A, B,\) and \(C\)), \texttt{n} will have three elements, which PYTHON refers to as \texttt{n[0]}, \texttt{n[1]}, and \texttt{n[2]}. If we assign the number of atoms of species \(A\) to the first element, the number of atoms of species \(B\) to the second element, and the number of atoms of species \(C\) to the third element, we have the associations \(A \mapsto n[0]\), \(B \mapsto n[1]\), and \(C \mapsto n[2]\). Rewriting the differential equations Eq. (11.19) in terms of the elements of \texttt{n}, we have the system of equations

\[
\frac{d}{dt}n[0] = -k_A n[0]; \quad \frac{d}{dt}n[1] = k_A n[0] - k_B n[1]; \quad \frac{d}{dt}n[2] = k_B n[1] \tag{11.101}
\]

Now we are ready to construct the function .py-file describing this system of differential equations. Respecting the general structure described in Section 11.10.3, we would compose the definition

```python
def decay( n, t):
    # DECAY: Returns derivatives for chain radioactive decay.
    # The function DECAY defines the rate equations for a three
    # species radioactive decay sequence such as
    #
    # n[0] --kA--> n[1] --kB--> n[2]
    #
    kA = 0.1; kB = 0.1
    derivs = [ -kA*n[0], kA*n[0]-kB*n[1], kB*n[1] ]
    return derivs
```

Here, the first line names the function \texttt{decay}, stipulates that the values of the dependent variables will be supplied in the \texttt{list n}, and stipulates that the value of the the independent variable will be supplied in the \texttt{scalar t}.\footnote{For this first example, we hard code the values of the parameters \texttt{kA} and \texttt{kB} in the function definition. Later in this section we will explain how to set these parameters in other ways.} Respectively, the two executable lines define the values of the two parameters and assign to the value of the function a \texttt{list}—commas separating entries—whose elements are the derivatives of the dependent variables as determined by the right hand sides of Eqs. (11.101).

With the function of the previous paragraph stored in the file \texttt{decay.py} in the default directory or in a directory in PYTHON’s search path,\footnote{See Sections 5.16.1 and 5.16.2 for information about the default directory and the search path.} we would produce a solution for the differential equations it defines with the statements
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint
execfile( 'decay.py' )
or exec(open( 'decay.py' ).read() )
t = np.linspace(0.0, 50.0, 401)
ic = \[1000.0, 0.0, 0.0 \]
n = odeint( decay, ic, t )

The solutions will be generated at the times in \( t \) with the solution at each time placed in the corresponding row of \( n \). We have accepted the default tolerances. (See Section 11.10.9.)

Once the solution is in hand in \( n \) and \( t \), we could plot the solution with the simple statement
\[
\text{plt.plot}( t, n ),
\]
which would plot the three columns of \( n \) in a single window, using a different color for each column. For purposes of black and white display on the page, is it better to use the statements
\[
\text{plt.plot}( t, n[:,0], '-k', linewidth=3 )
\text{plt.plot}( t, n[:,1], ':k', linewidth=3 )
\text{plt.plot}( t, n[:,2], '--k', linewidth=3 )
\text{plt.title}( 'Three-Species Chain Decay', fontsize=16 )
\text{plt.xlabel}( 'Time', fontsize=14 )
\text{plt.ylabel}( 'Number of Atoms Present', fontsize=14 )
\text{plt.tick_params(labelsize=12)}
\text{plt.show()}
\]

which will distinguish the three variables with line styles rather than color. The resulting graph—Fig. 11.12—shows the time variation of the number of atoms of each species.

Almost always, we will want to explore the dependence of the solution on the values of any parameters, e.g., \( k_A \) and \( k_B \) in the example of this section. Such explorations would be much easier if the parameters could be set before \( \text{odeint} \) is called and be known inside the function that \( \text{odeint} \) calls to determine the derivatives. Conveniently, \( \text{odeint} \) provides the keyword \text{args} for just this purpose.\(^{45}\)

To modify the above example, we would start by editing the function \text{decay} to be

\(^{45}\)We could also exploit global variables (see Section 5.7.3), but the route here described avoids inadvertent conflict of variable names with variables elsewhere in, particularly, a long program.
def decaymod( n, t, kA, kB ):
    # DECAY: Returns derivatives for chain radioactive decay.
    # The function DECAY defines the rate equations for a three
    # species radioactive decay sequence such as
    #
    # n[0] --kA--> n[1] --kB--> n[2]
    
    derivs = [ -kA*n[0], kA*n[0]-kB*n[1], kB*n[1] ]
    return derivs

Specifically, we have added the two parameters as additional arguments to the function. Invocation of odeint by the main program then is modified from the above as follows:

import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint
execfile( 'decaymod.py' )
    or exec(open( 'decaymod.py' ).read() )
t = np.linspace(0.0, 50.0, 401)
ic = [1000.0, 0.0, 0.0 ]
kA = 0.1; kB = 0.1;
n = odeint( decaymod, ic, t, args=(kA,kB) )

We have invoked the keyword args—a tuple—to odeint to provide the parameters that odeint simply passes on to the function it calls (here, decaymod.py). To explore other values for the parameters, we simply reset the value of kA and kB and re-execute the following statements in the sequence immediately above.

Figure 11.12: Solution of chain radioactive decay via odeint with a tolerance of about $10^{-8}$ absolute, $10^{-8}$ relative (see Section 11.10.9). The solid, dotted and dashed lines show $A(t)$, $B(t)$, and $C(t)$, respectively.
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 $\texttt{odeint}$ with the (default) tolerance of $\approx 10^{-8}$ (see Section 11.10.9). 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 seconds. 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.

## 11.10.5 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 py-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 $\pi \mapsto y[0]$ and $\tau \mapsto y[1]$. In these terms, the differential equations become

$$\frac{d}{dt} y[0] = y[1] \quad ; \quad \frac{d}{dt} y[1] = -y[0] - \beta y[1] \quad (11.102)$$

Then, using the symbol $\beta$ for $\sqrt{\beta}$, we create the py-file

```python
def damposc( y, t, bb ):
    # DAMPOSC: returns derivatives for damped 1D harmonic oscillator
    # The function DAMPOSC describes the nondimensionalized equations of
    # motion for a damped, one dimensional harmonic oscillator. $\beta$ is a
    # dimensionless parameter and has a value of $\beta=b/\sqrt{m*k}$, where $b$
    # is the damping, $m$ is the mass and $k$ is the spring constant.
    # $\beta$ will be passed to damposc through odeint with the keyword args.

dervs = [ y[1], -y[0] - bb*y[1] ];
return derivs
```

Defining the differential equations. Storing this file in the default directory with the name `damposc.py`, we explore the system for two different damping constants with the statements

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint
execfile( 'damposc.py' )
or exec(open( 'damposc.py' ).read() )
t1 = np.linspace(0.0, 30.0, 151)
ic = [1.0, 0.0 ]
bb=0.0;
y1 = odeint( damposc, ic, t1, args=(bb,) )
```

Remember that a tuple with a single component requires a comma following that component.

46 Remember that a tuple with a single component requires a comma following that component.
Set small damping.

Solve equations with same ic's and times as before but putting solution in y2.

Finally, we produce graphs showing the position and velocity as functions of time and also the trajectory in the phase plane (v versus x) for each value of the damping constant with the statements

```python
bb=0.25
y2 = odeint( damposc, ic, t1, args=(bb,) )
```

The resulting graphs are shown in Fig. 11.13.

## 11.10.6 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 system of two second-order equations as a quartet of first-order equations. Thus, we view the equations as the system

\[
\frac{d\vec{r}}{dt} = \vec{v} \\
\frac{d\vec{v}}{dt} = -\frac{\vec{r}}{(x^2 + y^2)^{1.5}} \\
\frac{d\vec{y}}{dt} = \vec{v} \\
\frac{d\vec{v}}{dt} = -\frac{\vec{y}}{(x^2 + y^2)^{1.5}}
\]  

(11.103)

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 py-file, we set \( \vec{r}, \vec{v}, \vec{y}, \) and \( \vec{v} \) 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 py-file `planet.py` defining the above equations is listed in Table 11.8. 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 PYTHON and pursue solutions to the planetary problem. We use `odeint` again and accept its default tolerances (see Section 11.10.9), 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

```python
fig, ( (ax1,ax2), (ax3,ax4) ) = plt.subplots(2,2)
fig.subplots_adjust(wspace=0.4, hspace=0.4)

ax1.plot(t1, y1[:,0], 'k-', linewidth=2)
ax1.plot(t1, y1[:,1], 'k--', linewidth=2)
ax1.set_title( 'Undamped Harmonic Oscillator' )
ax1.set_xlabel('Time')

ax2.plot(t1, y2[:,0], 'k-', linewidth=2)
ax2.plot(t1, y2[:,1], 'k--', linewidth=2)
ax2.set_title( 'Damped Harmonic Oscillator' )
ax2.set_xlabel('Time')

ax3.plot(y1[:,0], y1[:,1], 'k-', linewidth=2)
ax3.set_xlabel('Position'); ax3.set_ylabel('Velocity')

ax4.plot(y2[:,0], y2[:,1], 'k-', linewidth=2)
ax4.set_xlabel('Position'); ax3.set_ylabel('Velocity')

plt.show()
```
Figure 11.13: Damped and undamped harmonic oscillators. The upper two graphs show position (solid line) and velocity (dashed line) as functions of time; the lower two graphs show the phase plane plot. Notice the difference between the phase plane orbits.

Table 11.8: The PYTHON function \texttt{planet.py}.

```python
def planet( x, t, b ):
    # PLANET: returns the derivatives for planet in field of sun
    # The function PLANET describes the 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 vector of dependent
    # variables are X-position, X-velocity, Y-position, Y-velocity.
    # The parameter $b$ will be passed to planet through odeint with
    # the keyword args.
    temp = ( x[0]**2 + x[2]**2 )**b
    derivs = [ x[1], -x[0]/temp, x[3], -x[2]/temp ]
    return derivs
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint
execfile( 'planet.py' ) or exec(open( 'planet.py' ).read() )
b=1.5

t = np.linspace(0.0, 60.0, 241 )
ic1 = [ 4.0, 0.0, 0.0, 0.5 ]
x1 = odeint( planet, ic1, t, args=(b,) )
ic2 = [ 4.0, 0.0, 0.0, 0.3 ]
x2 = odeint( planet, ic2, t, args=(b,) )

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. For brevity, we have chosen to omit the statements creating these graphs but note the need to invoke

- plt.axis('square') or plt.axes().set_aspect('equal') in the graphs of $y$ versus $x$ so both axes will be scaled the same way,
- plt.tick_params(labelsize=14) to enlarge the tick labels on each axis, and
- the keyword fontsize to enlarge the labels on all axes.

Except for these embellishments, the statements generating Fig. 11.14 should be constructed with little difficulty.

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

$$
E_1 = 0.5 \times (x1[:,1]**2+x1[:,3]**2) - 1.0/\text{np.sqrt}(x1[:,0]**2+x1[:,2]**2)
$$

$$
L_1 = x1[:,0]*x1[:,3] - x1[:,2]*x1[:,1]
$$

$$
E_2 = 0.5 \times (x2[:,1]**2+x2[:,3]**2) - 1.0/\text{np.sqrt}(x2[:,0]**2+x2[:,2]**2)
$$

$$
L_2 = x2[:,0]*x2[:,3] - x2[:,2]*x2[:,1]
$$

will calculate these quantities. Then, statements like

```python
print(E1); print(L1); print(E2), print(L2)
```

will display the resulting values. We find—see your own output—that, rounded to five digits after the decimal point, $E_1 = -0.12500, L_1 = 2.00000, E_2 = -0.20500, and L_2 = 1.20000$ at every point throughout the entire solutions. 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 PYTHON’s for loop to produce a variety of open orbits for a planet started in different initial positions by executing the statements

```
b=1.5
y0 = [-4.0, -3.0, -2.0, -1.0, 1.0, 2.0, 3.0, 4.0]
plt.xlim(-4.0,4.0); plt.ylim(-4.0,4.0)
plt.xlabel( 'X-Position', fontsize=16 )
plt.ylabel( 'Y-Position', fontsize=16 )
plt.title( 'Open Orbits', fontsize=20 )
t = np.linspace(0.0, 60.0, 241)
```
for i in np.arange(0,8):
    ic=[-4.0, 1.0, y0[i], 0.0]
    x = odeint( planet, ic, t, args=(b,) )
    plt.plot( x[:,0], x[:,2], linewidth=2.0, color='black' )
    plt.plot( [-4.0, 0.0], [0.0, 0.0], linewidth=2.0, color='black' )
plt.grid(color='black')
plt.show()

Note that, for an orbit that heads directly towards the (attractive) force center \( y0 = 0 \), the nu-
numerical solution encounters a divergence, the planet collides with the force center, and (presumably) disappears. That one “orbit” has been omitted from the loop but added with the first statement after the loop. Figure 11.15 shows the resulting output.

11.10.7 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

\[
\frac{df}{dx} = g ; \quad \frac{dg}{dx} = -k^2 f ; \quad f(0) = 0 ; \quad f(\ell) = 0
\]  

(11.104)

Thus, setting up the correspondences \( f \mapsto f[0] \) and \( g \mapsto f[1] \), we might define this system for PYTHON with the py-file
def stdwaves( f, x, k ):
    # STDWAVES: returns derivatives for standing waves in string
    # The function STDWAVES defines the basic equations for describing
    # standing waves in a string. The parameter k is passed via the
    # keyword args to odeint.
    derivs = [ f[1], -k**2*f[0] ]
    return derivs

which we store in the default directory with the name stdwaves.py.

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 \( df(0)/dx = g(0) \) 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 the equation in dimensionless form with \( \ell \) chosen as the unit of length). Using odeint, we would invoke the statements

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint
execfile( 'stdwaves.py' ) or exec(open( 'stdwaves.py' ).read() )
k = 1.0
ic = [0.0, 1.0]
x = np.linspace(0.0,1.0, 51 )
f = odeint( stdwaves, ic, x, args=(k,) )
plt.plot( x, f[:,0], linewidth=2.0, color='black' )
plt.ylim([-1.0, 1.0])
plt.tick_params(labelsize=12)
```

The result is the upper solid line in Fig. 11.16. We are accepting that the default tolerance (Section 11.10.9) is adequate for this and 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 \( g(0) = 1.0 \) 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

```python
k = 2.0
f = odeint( stdwaves, ic, x, args=(k,) )
plt.plot( x, f[:,0], linewidth=2.0, color='black', linestyle=':' )
k = 3.0
f = odeint( stdwaves, ic, x, args=(k,) )
plt.plot( x, f[:,0], linewidth=2.0, color='black', linestyle='--' )
k = 4.0
f = odeint( stdwaves, ic, x, args=(k,) );
plt.plot( x, f[:,0], linewidth=2.0, color='black' )
plt.grid()
```

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 \( k \).
Figure 11.16: Shooting method to solve for standing waves in a string. The upper solid, dotted, dashed, and lower solid curves correspond to \( k = 1.0, 2.0, 3.0, \) and \( 4.0, \) respectively.

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

```python
k = np.linspace(0.0, 20.0, 101)
ic = [0.0, 1.0]
x = np.linspace(0.0, 1.0, 51)
soln = []

for i in np.arange(0, 101):
    f = odeint(stdwaves, ic, x, args=(k[i],))
soln = np.append(soln, f[50,0])

plt.plot(k, soln, linewidth=2.0, color='black')
plt.xlabel('k', fontsize=14)
plt.ylabel('f(1)', fontsize=14)
plt.tick_params(labelsize=12)
plt.grid()
```

Here, we

- Establish in \( k \) values that will be passed to \( k \), one at a time.
11.10. SOLVING ODES NUMERICALLY WITH PYTHON

Figure 11.17: Graph of $f(1)$ versus $k$ for standing waves in a string.

- Set the initial conditions.
- Set the desired $x$ coordinates.
- Initialize a variable for the solution.
- Execute a loop that (a) invokes `odeint` to solve the equations for a value of $k$ and (b) stores in `soln[i]` the solution at $x = 1$ for the current value of $k$.
- Plot the resulting values.

Note that the loop requests a fair bit of computation and may take awhile to execute.

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 $0 \leq k \leq 20$. The first six, however, suggest (correctly) that the solutions are equally spaced in $k$.

11.10.8 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\(^{47}\) $0 \leq y < \infty$. 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

\(^{47}\) We here use the symbol $y$ for what we earlier called $x$. 
def qmshm( psi, y, ee ):
    # QMSHM: returns the derivatives for the quantum harmonic oscillator
    # QMSHM defines the dimensionless Schroedinger equation for the quantum
    # simple harmonic oscillator, which is
    #
    # 2
    # d psi 2
    # ----- + (2 ee - y ) psi = 0
    # dy
    #
    # Entries are psi and d psi/dy, and initial conditions will thus be
    # [psi0, dpsi0]. The parameter ee represents a dimensionless energy
    # and is passed to qmshm via the args keyword in odeint.
    derivs = [ psi[1], -(2*ee-y**2)*psi[0] ]
    return derivs

equations we must solve then become

\[
\frac{d}{dy}\psi[0] = \psi[1] ; \quad \frac{d}{dy}\psi[1] = -(2\epsilon - y^2)\psi[0] \tag{11.105}
\]

and a suitable py-file qmshm.py defining these equations for PYTHON's solvers is listed in Table 11.9. With this file stored in the default directory with the name qmshm.py, 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

import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint
execfile( 'qmshm.py' ) or exec(open( 'qmshm.py' ).read() )
ic = [ 1.0, 0.0 ]; ee = 1.0
y = np.linspace(0.0, 4.0, 41 )
psi = odeint( qmshm, ic, y, args=(ee, ) )
plt.plot( y, psi[:,0] )
plt.xlim( [0.0,4.0] ); plt.ylim( [-2.0,2.0] )
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

```python
e = [ 0.3, 0.4, 0.5, 0.6, 0.7, ]
for ee in e:
    psi = odeint( qmshm, ic, y, args=(ee,) )
    plt.plot( y, psi[:,0], linewidth=2.0, color='black' )
    plt.xlim( [0.0, 4.0] ); plt.ylim( [-2.0, 2.0 ] )
    plt.plot( [0.0,4.0], [0.0, 0.0], color='black' )
    plt.text( 2.0, 1.7, '0.3', fontsize=16 )
    plt.text( 2.8, 1.2, '0.4', fontsize=16 )
    plt.text( 3.5, 0.2, '0.5', fontsize=16 )
    plt.text( 2.8, -1.0, '0.6', fontsize=16 )
    plt.text( 2.2, -1.8, '0.7', fontsize=16 )
    plt.xlabel( '$y$', fontsize=16 )
    plt.ylabel( '$\psi(y)$', fontsize=16 )
    plt.tick_params(labelsize=12)
    plt.grid()
```

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 $\epsilon = 0.5$.

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

```python
e = [ 1.0, 1.5, 2.0, 2.5, 3.0, 3.5 ]
```
to set the values of $\epsilon$ and the lines

```python
plt.text( 2.1, -1.2, '1.0', fontsize=16 )
plt.text( 1.7, -1.7, '1.5', fontsize=16 )
plt.text( 2.7, -1.8, '2.0', fontsize=16 )
plt.text( 3.5, -0.3, '2.5', fontsize=16 )
plt.text( 2.7, -1.8, '2.0', fontsize=16 )
plt.plot( [0.0,4.0], [0.0, 0.0], color='black' )
plt.xlabel( '$y$', fontsize=16 )
plt.ylabel( '$\psi(y)$', fontsize=16 )
plt.tick_params(labelsize=12)
plt.grid()
```

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

```python
e = [ 3.0, 3.5, 4.0, 4.5, 5.0, 5.5 ]
for ee in e:
    psi = odeint( qmshm, ic, y, args=(ee,) )
    plt.plot( y, psi[:,0], linewidth=2.0, color='black' )
    plt.xlim( [0.0, 4.0] ); plt.ylim( [-2.0, 2.0 ] )
    plt.plot( [0.0,4.0], [0.0, 0.0], color='black' )
    plt.text( 2.0, 1.7, '3.0', fontsize=16 )
    plt.text( 2.8, 1.2, '3.5', fontsize=16 )
    plt.text( 3.5, 0.2, '4.0', fontsize=16 )
    plt.text( 2.8, -1.0, '4.5', fontsize=16 )
    plt.text( 2.2, -1.8, '5.0', fontsize=16 )
    plt.text( 3.0, 1.2, '5.5', fontsize=16 )
    plt.xlabel( '$y$', fontsize=16 )
    plt.ylabel( '$\psi(y)$', fontsize=16 )
    plt.tick_params(labelsize=12)
    plt.grid()
```

which, if we now allow $\epsilon$ to increase beyond the value 5.0, 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

```python
e = [ 6.0, 6.5, 7.0, 7.5, 8.0, 8.5 ]
```
to set the values of $\epsilon$ and the lines

```python
plt.text( 6.1, -1.2, '6.0', fontsize=16 )
plt.text( 5.7, -1.7, '6.5', fontsize=16 )
plt.text( 6.7, -1.8, '7.0', fontsize=16 )
plt.text( 7.5, -0.3, '7.5', fontsize=16 )
plt.text( 6.7, -1.8, '7.0', fontsize=16 )
plt.plot( [0.0,4.0], [0.0, 0.0], color='black' )
plt.xlabel( '$y$', fontsize=16 )
plt.ylabel( '$\psi(y)$', fontsize=16 )
plt.tick_params(labelsize=12)
plt.grid()
```

produces the graph shown in Fig. 11.20. From this graph, we conclude that $\epsilon = 7.5$ yields another solution that goes to zero at infinity, and we argue that another allowed energy corresponds to the value $\epsilon = 7.5$. We might even be tempted to speculate—correctly as it turns out—that additional allowed energies for odd states will correspond to the values $\epsilon = 9.5, 11.5, 13.5, \ldots$.
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).

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.
Figure 11.20: The solution to the Schrödinger equation for an odd state of the quantum harmonic oscillator when $\epsilon$ has the indicated values.

```python
ic = [ 0.0, 1.0 ]
e = [ 1.3, 1.4, 1.5, 1.6, 1.7 ]

Then we execute the statements

```python
for ee in e:
    psi = odeint( qmshm, ic, y, args=(ee, )
    plt.plot( y, psi[:,0], linewidth=2.0, color='black' )
```

to generate the graph in Fig. 11.20 and the statements

```python
plt.xlim( [0.0, 4.0] ); plt.ylim( [-2.0, 2.0 ]
plt.text( 2.7, 1.7, '1.3', fontsize=16 ); plt.text( 3.5, 1.5, '1.4', fontsize=16 );
plt.text( 3.5, 0.1, '1.5', fontsize=16 ); plt.text( 3.4, -1.0, '1.6', fontsize=16 );
plt.text( 2.8, -1.5, '1.7', fontsize=16 );
plt.xlabel( '$y$', fontsize=16 )
plt.ylabel( '$\psi(y)$', fontsize=16 )
plt.tick_params(labelsize=12)
plt.grid()
```

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 \( E = \epsilon \bar{h} \omega \), we infer that the energies of the quantum harmonic oscillator are given by
\[
E_n = \left( n + \frac{1}{2} \right) \hbar \omega
\] (11.106)
though we have direct evidence for this (correct) conclusion only for the lowest three states.

11.10.9 Keywords that Modify odeint

When the default values of the keywords that control the detailed action of odeint are not appropriate to the task at hand, these keywords can be given different values. The available keywords with the default values are

- **atol** and **rtol**, which specify the maximum absolute and relative errors, respectively, in the dependent variables. These keywords may be scalars, in which case the values apply to all dependent variables, or they may be a lists, in which case each component specifies the tolerances for the corresponding dependent variable. Both default to \( 1.49012 \times 10^{-8} \).
- **h0**, which specifies a starting value for the step size, which is then adjusted as the adaptive procedure works out the solution. The default is 0.0, which tells PYTHON to make its own choice.
- **hmin**, which specifies the minimum step size allowed in the adaptive process of finding the solution. The default is 0.0, which tells PYTHON to make its own choice.
- **hmax**, which specifies the largest step size the solver is to use as the adaptive procedure works out the solution. The default is 0.0, which tells PYTHON to make its own choice.
- **mxstep**, which specifies the maximum number of steps allowed for each integration point in the solution. The default is 0.0, which tells PYTHON to make its own choice.

The two tolerances are both taken into account as PYTHON assesses convergence. Specifically, PYTHON seeks to keep the absolute value of the local error \( e \) in each dependent variable such that \( \frac{e}{\text{ewt}} \leq 1.0 \) in each of the returned values of the dependent variables, where
\[
|\text{ewt}| = \text{rtol} \cdot |y| + \text{atol}
\]
Here, \( y \) is the returned value of the solution at every step along the way. Typically, when the solution is close to zero, the absolute tolerance provides the controlling criterion while, when the solution is well away from zero, the relative tolerance may dominate over the absolute tolerance. Either (but certainly not both) of these tolerances can be set to zero, in which case the other provides the controlling criterion throughout the solution. Be warned, however: if the absolute tolerance is set to zero, convergence may be difficult when the solution itself is close to zero (since a small fraction of zero—relative tolerance—may result in excessive computation).

To this point, we have accepted the default tolerances in each of the sample solutions. If, for example, in the solution of the problem in radioactive decay, we had wanted to change the absolute tolerance to 5.0 and the relative tolerance to 10.0%, we would have replaced statements like
\[
n = \text{odeint}( \text{decay}, \text{ic}, \text{t} )
\]
with
n = odeint( decay, ic, t, atol=5.0, rtol=0.1 )

Confirmation that the solution generated by this coding and the solution generated with the default tolerances differ by very little is left as an exercise for the reader.

Further detail about the options that affect the behavior of `odeint` can be found in the PYTHON manuals.

### 11.10.10 py-files in the Public Library

The following py-files are found in the directory `$HEAD/python` (and may also at some sites have been placed in a directory in PYTHON’s default search path) and can be used to explore the systems they describe. Further information about the file `decay.py`, for example, can be obtained either with the command  

```python
print( open('decay.py').read() )
```

to PYTHON or with whatever command to your operating system lists the file on the screen. The first of these commands instructs PYTHON to open the file `decay.py` and display the entire file, including any comments in the file; the second command, which must be executed in a command window to the operating system (not within PYTHON), will display the entire file on the screen. Many of the functions defined by these files involve parameters that must be supplied with the keyword `args` to `odeint`; examine the comments in the file itself to determine the proper specification of these variables.

- **apollo.py**: describes a satellite in the gravitational attraction of two fixed suns of equal mass.
- **dampharm.py**: describes the one-dimensional damped, unforced harmonic oscillator using dimensional variables.
- **damposc.py**: describes the one-dimensional, damped, unforced harmonic oscillator using dimensionless variables.
- **decay.py**: describes three-species radioactive decay.
- **decaymod.py**: describes three-species radioactive decay using global variables to communicate parameters to the function.
- **drvnosc.py**: describes the one-dimensional, damped, forced harmonic oscillator using dimensional variables.
- **drvnpend.py**: describes the forced, damped simple pendulum in dimensionless units.
- **henon.py**: describes the Hénon-Heiles oscillator.
- **largamp.py**: describes the undamped, unforced large amplitude pendulum in dimensionless units.
- **lorenz.py**: describes the Lorenz attractor.
- **onedshm.py**: describes the one-dimensional, undamped, unforced harmonic oscillator using dimensional variables.

---


49Specification of a full path will be necessary if the file is not in a directory in the default directory of PYTHON’s search path.

50In UNIX and windows, one possible command would be `more $HEAD/python/decay.py`; see the *Local Guide*. 

• **planet.py**: 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.

• **qmshm.py**: describes the dimensionless quantum harmonic oscillator.

• **rossler.py**: describes the Rössler attractor.

• **stdwaves.py**, which describes the behavior of standing waves in a string.

• **twodshm.py**: describes the two-dimensional, undamped, unforced harmonic oscillator using dimensionless variables.

• **twooscil.py**: describes a system of two equal masses between two walls, all connected by three springs. Dimensionless units are used.

• **vandpol.py**: describes the Van der Pol oscillator.

### 11.10.11 Other ODE Solvers in PYTHON

In addition to the solver **odeint**, the **scipy.integrate** module includes two other ODE solvers, specifically

- **scipy.integrate.ode**,\(^{51}\) which is a bit more complicated to use than **odeint**. In particular,
  - The order of the first two arguments in the function defining the differential equation(s) is reversed *(IndVar, DepVars)* from the order *(DepVars, IndVar)* proper for **odeint**. Consequently functions used with **odeint** must be recast to work with **ode**.
  - Rather than being provided by keywords as in the call to **odeint**, specification of various options (tolerances, step sizes, ...) are set by invoking **scipy.integrate.ode.set_integrator**.
  - Different methods (Adams, backward differentiation formulas, lsoda, Runge-Kutta methods) of integration can be explicitly selected.

- **scipy.integrate.solve_ivp**,\(^ {52}\) which functions more like **odeint** but
  - Like **ode** requires a function defining the differential equations with the variables in the order *(IndVar, DepVars)*.
  - Offers selection of a variety of methods, including fourth-fifth order Runge-Kutta, second-third order Runge Kutta, backward differentiation, and lsoda.
  - Like **odeint**, accepts keywords to control various options.
  - Properly used, can solve complex ODEs.

### 11.13 Solving ODEs Numerically with Mathematica

**Mathematica**’s command **NDSolve** provides a means to solve ordinary differential equations numerically. A full statement using **NDSolve** will have the form

```mathematica
NDSolve[ { ODE1, ODE2, ..., IC1, IC2, ...}, { DVar1, DVar2, ...},
           { IVar, IVarMin, IVarMax } ]
```


\(^{52}\) See [docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.solve_ivp.html](https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.solve_ivp.html) for details.
where ODE and IC are the differential equations and initial conditions, DVar represents a dependent variable, and IVar, IVarMin, and IVarMax are the independent variable and the lower and upper limits on the range in which the solution is to be found. In this case, both the differential equations and the initial conditions must be specified; Mathematica will not introduce undetermined integration constants. Further, all parameters in the equations must be given explicit numerical values before NDSolve is executed. When invoked, NDSolve returns not a solution but an interpolation function that, once a specific numerical value has been assigned to the independent variable, can be used to generate a numerical value for the solution at the specified value of the independent variable. Thus, for example, to address the problem of chain radioactive decay that we have already solved symbolically in Section 11.5.4, we would request Mathematica to generate the interpolation function by invoking the statements

\begin{verbatim}
In[1]:= eqA = D[AA[t], t] == -kA*AA[t]; Enter equations, suppressing display.
In[2]:= eqB = D[BB[t], t] == kA*AA[t] - kB*BB[t];
In[3]:= eqC = D[CC[t], t] == kB*BB[t];
In[4]:= kA = 0.1; kB = 0.1;
In[5]:= soln = NDSolve[{eqA, eqB, eqC, AA[0]==1000, BB[0]==0, CC[0]==0},
{AA, BB, CC}, {t, 0, 50}]
\end{verbatim}

Out[5]= {{AA → InterpolatingFunction[Graphic], BB → InterpolatingFunction[Graphic],
CC → InterpolatingFunction[Graphic]}}

Then, after the interpolating function has been created, the solution at any particular point can be found with statements like

\begin{verbatim}
In[6]:= {AA[0.0], BB[0.0], CC[0.0] } /. soln
Out[6]= {{1000., 0., 0.}}
In[7]:= {AA[0.1], BB[0.1], CC[0.1] } /. soln
Out[7]= {{990.05, 9.9005, 0.0496679}}
\end{verbatim}

Note the form of the solution, which is presented as a list of lists, though this particular solution has only one list in the outer list. In that (one) list, the elements are the numerical values of the solution at the selected value of the independent variable—time in this case.

Should we wish a graph of the solution over some range of the independent variable, we would invoke the command Plot but use the command Evaluate to determine the values to be plotted. A graph similar to Fig. 11.9 for this decay would, for example, be displayed by the statement

\begin{verbatim}
In[8]:= Plot[Evaluate[{AA[t], BB[t], CC[t]} /. soln, {t, 0.0, 50.0}],
PlotStyle -> Thickness[0.01], AxesLabel -> {"t", "A,B,C"} ]
\end{verbatim}

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 Mathematica 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 Mathematica manuals. While the defaults adopted for controlling accuracy and for protecting against infinite loops are often acceptable, the command NDSolve nonetheless possesses several options that give us the ability to override these defaults. Among these options are

\begin{verbatim}
53Because C is a protected variable in Mathematica, the dependent variables here are named AA, BB, and CC rather than A, B, and C.
54The inclusion identified here with the word ‘Graphic’ is a box that has a small graph of the function as well as an indication of the domain of the solution and the character of the output. Clicking ML on the ‘+’ sign in that graphic brings up information about other features of the solution.
\end{verbatim}
• **WorkingPrecision** (default $\text{MachinePrecision}$, often 16), which specifies the number of digits to be used for *internal* computations.

• **AccuracyGoal** (default usually figured as half the value of WorkingPrecision), which specifies the *absolute* accuracy to be sought by stipulating the number of digits to be provided in the solution. **AccuracyGoal** should not exceed **WorkingPrecision**.

• **PrecisionGoal** (default usually figured as half the value of WorkingPrecision), which specifies the *relative* accuracy sought in the final result. **PrecisionGoal** should not exceed **WorkingPrecision**.

  *Note*: With AccuracyGoal set to $a$ and PrecisionGoal set to $p$, **Mathematica** strives for an absolute accuracy in a value $x$ to be less than $10^{-a} + 10^{-p} |x|$. 

• **MaxSteps** (default automatic), which specifies the maximum number of steps to take in generating the solution. This option provides a means to stop the iterations even if goals on accuracy and precision have not been met.

• **MaxStepSize** (default automatic), which limits the size of the largest steps to be taken in generating the solution. This option provides a means to specify, for example, a maximum step size smaller than **Mathematica** determines to be adequate.

Remember, too, that the statement Options[NDSolve] will display all options and their current default values for the indicated command.

If we want solutions at a specific set of values of the independent variable, we could exploit the command Table. For example, the statement

```mathematica
In[9]:= tbl = Table[{t, AA[t], BB[t], CC[t]} /. soln, {t, 0, 20, 5} ]
```

produces a table named `tbl` whose $i$-th element contains four values (the time and the values of $A$, $B$, and $C$ at that time). This output can then be manipulated using any of **Mathematica**’s resources for working with tables.

### 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 $\text{HEAD/fortran}$, where (as defined in the Local Guide) $\text{HEAD}$ 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.
11.14. SOLVING ODES NUMERICALLY WITH FORTRAN

Table 11.10: First part of FORTRAN coding for decay.f.

```fortran
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. (11.79)–(11.81) 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 11.10.\textsuperscript{55,56} 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

```fortran
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 11.11. 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 statements\textsuperscript{57}

\textsuperscript{55}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.

\textsuperscript{56}Because we are using implicit data typing, we use \texttt{ak} and \texttt{bk} rather than \texttt{ka} and \texttt{kb} for the decay constants to avoid implying to FORTRAN that these quantities are to be treated as integers.

\textsuperscript{57}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 \texttt{./} is necessary in UNIX but may not be necessary with other operating systems. Details will be found in the \textit{Local Guide}. 

Table 11.11: 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
./decay.xf
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.f.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>1000.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>0.25</td>
<td>975.000</td>
<td>25.000</td>
<td>0.000</td>
</tr>
<tr>
<td>0.50</td>
<td>950.625</td>
<td>48.750</td>
<td>0.625</td>
</tr>
<tr>
<td>0.75</td>
<td>926.859</td>
<td>71.297</td>
<td>1.844</td>
</tr>
<tr>
<td>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 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 hierarchy are routines—called stepper routines—which use an algorithm routine to advance the solution by one step but do so adaptively. Each stepper routine calls an algorithm routine repeatedly with different step sizes until its assessment of the error (per step) satisfies a user-specified tolerance. The stepper routine that uses rkck.f is named rkqs.f; the stepper routine that uses mmid.f is 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 hierarchy. That is, we shall be constructing routines—called driver routines or, perhaps more accurately in our case, driver programs—which use a stepper algorithm to generate a solution over a given total interval.58,59

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—also 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 be60

```
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
```

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.f, which is called with a statement of the form

```
CALL RK4( YS, DYDT, NEQS, T, DT, YF, DRVS )
```

Here ys and dydt are vectors containing the values of the dependent variables and their derivatives at time t, neqs is the number of equations in the system, dt is the amount by which the independent variable is to be advanced, yf is a vector that receives the values of the dependent variables at time t+dt, and drvs, which is a string value that is not enclosed in quotation marks, is the name of the subroutine that rk4.f is to invoke to calculate the derivatives—the subroutine structured as

58 Alternatively, we might exploit the driver routines rkdumb.f and odeint.f in the Numerical Recipes library.
59 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_f/demo/src. The sample program corresponding to a particular routine is named by prefixing the character x to the routine name, e.g., xrk4.f for rk4.f.
60 Again, to avoid having FORTRAN interpret what should be floating point variables as integers, we name the vector for the dependent variables an and the parameters ak and bk.
described in the last paragraph. Note (1) that *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 *rk4.f*; (2) *yf* can be the same vector as *ys*, so the output produced by *rk4.f* can be stored on top its input; and (3) *rk4.f* does not increment *t*, so an explicit incrementation will be necessary before a subsequent call to *rk4.f* in a loop.

A possible driving program to use subroutine *der* (which we have written) and subroutine *rk4.f* (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

```fortran
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
```

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

```fortran
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 )
```

Finally, we execute a loop in which, for each pass, the derivatives at the start of the current interval are evaluated, *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

```fortran
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 )
```

A full listing of this program, which is named *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 *rk4.f*, we obtain the actual solution and store it in the file *decayrk4_f.dat* by

1. Creating the driving program *decayrk4.f* and the subroutine *der* using an available text editor (or copying the files from the directory $HEAD/fortran$).

---

61 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.
2. Copying the recipe $\texttt{rk4.f}$ from the directory $\texttt{SNHEAD/recipes_f/recipes}$ to the default directory.

3. Compiling and linking the executable module with the statement

\begin{verbatim}
f77 -o decayrk4.xf decayrk4.f rk4.f
\end{verbatim}

4. Running the program with the statement $\texttt{./decayrk4.xf}$.

The first and last few lines in the file $\texttt{decayrk4.dat}$ 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  951.22 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 $y$ and $dydt$ are vectors containing the values of the dependent variables and their derivatives at time $t$, $\texttt{neqs}$ is the number of equations in the system, $\texttt{dttry}$ is the amount by which the independent variable is to be advanced, $\texttt{rtol}$ is the desired relative tolerance, $\texttt{ys}$ is a vector containing the values relative to which the tolerance is to be assessed, $\texttt{dtact}$ is the actual step size on which the adaptive process ultimately settled, $\texttt{dtnext}$ is the recommended step size for the next step in the solution, and $\texttt{drvs}$ 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 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 $y$ and $t$, 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 11.12. 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.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 11.12: 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
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. (11.79)–(11.81) 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:\footnote{Remember that C is case sensitive.}

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>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>49.50</td>
<td>7.083</td>
<td>35.063</td>
<td>957.854</td>
</tr>
<tr>
<td>49.75</td>
<td>6.909</td>
<td>34.370</td>
<td>958.722</td>
</tr>
<tr>
<td>50.00</td>
<td>6.738</td>
<td>33.690</td>
<td>959.572</td>
</tr>
</tbody>
</table>

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.
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 %.10f %.10f %.10f\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, "%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 11.13. 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

\[63\]

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.
cc -o decay.xc decay.c  // Create executable file decay.xc.
./decay.xc  // Execute program.
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>1000.00</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>0.25</td>
<td>975.000</td>
<td>25.000</td>
<td>0.000</td>
</tr>
<tr>
<td>0.50</td>
<td>950.625</td>
<td>48.750</td>
<td>0.625</td>
</tr>
<tr>
<td>0.75</td>
<td>926.859</td>
<td>71.297</td>
<td>1.844</td>
</tr>
<tr>
<td>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 recasting 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 hierarchy are routines—called *algorithm* routines—which use one or another basic algorithm to advance the solution to a system of ODEs by one step of specified size. These routines are not adaptive, and they make no attempt to assess the accuracy of their output. In this category, we shall focus on the routine *rk4.c*, which advances the solution one step by the fourth-order Runge-Kutta method, and the routine *rkck.c*, which advances the solution one step by the fifth-order Cash-Karp Runge-Kutta method. Other routines in this category, including *mmid.c* (which implements a modified midpoint method), are used in the same way.

One rung up in the Numerical Recipe hierarchy are routines—called *stepper* routines—which use an algorithm routine to advance the solution by one step but do so adaptively. Each stepper routine calls an algorithm routine repeatedly with different step sizes until its assessment of the error (per step) satisfies a user-specified tolerance. The stepper routine that uses *rkck.c* is named *rkqs.c*; the stepper routine that uses *mmid.c* is *bsstep.c*.

We can, of course, use both algorithm and stepper routines directly, and we shall confine ourselves to illustrating those uses. To generate a full solution over some extended interval of the independent variable, therefore, we will have to construct our own loops to execute the routines repeatedly. In doing so, we will be constructing for ourselves versions of programs that constitute the top rung in the Numerical Recipe hierarchy. That is, we shall be constructing routines—called *driver* routines or, perhaps more accurately in our case, *programs*—which use a stepper algorithm to generate a solution over a given total interval.\(^64,65\)

\(^64\) Alternatively, we might exploit the driver routines *rkdumb.c* and *odeint.c* in the Numerical Recipes library.

\(^65\) 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 $NRHEAD/recipes.c-ansi/demo/src$.\(^66\) 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*.\(^66\)
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). For our standard three-species radioactive decay, for example, we would have to define the two parameters $k_A$ and $k_B$ as global variables with the statement

```c
float kA, kB; /* For parameters */
```

placed before the definition of the subroutine defining the derivatives and before the statement `main()` that introduces the main program. Then, we might code the required subroutine—name it `der—with the statements

```c
void der( float t, float n[], float dndt[] )
{
    dndt[1] = -kA*n[1]; /* Calculate dA/dt */
    dndt[3] = kB*n[2]; /* Calculate dC/dt */
}
```

Second, we must create a driving program that defines appropriate variables, assigns values to all necessary parameters, calls the solver—usually in a loop—to calculate the actual solution, and generates a file containing the results of that calculation. We illustrate first the use of `rk4.c`, which is called with a statement of the form

```c
rk4( ys, dydt, neqs, t, dt, yf, drvs )
```

Here `ys` and `dydt` are vectors containing the values of the dependent variables and their derivatives at time `t`, `neqs` is the number of equations in the system, `dt` is the amount by which the independent variable is to be advanced, `yf` is a vector that receives the values of the dependent variables at time `t+dt`, and `drvs`, which is a string value that is not enclosed in quotation marks, is the name of the subroutine that `rk4.c` is to invoke to calculate the derivatives—the subroutine structured as described in the last paragraph. 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

```c
69
```

Details on the use of global variables in C are discussed in Section 9.7.4.

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.

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].`
float n[3+1], dn dt[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\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, 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, dn dt );
    rk4( n, dn dt, 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 );

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 11.14: First statements for C Program decayrkqs.c.

```c
float n[3+1], dndt[3+1]; /* For variables and derivatives */
float rel; /* For error control */
float t, dttry; /* For time, time steps */
kA = 0.1; kB = 0.1; /* Set parameters */
dttry = 0.25; /* Set time step */
n[1] = 1000.0; n[2] = 0.0; /* Initialize dependent variables */
n[3] = 0.0;
t = 0.0; /* Set initial time */
rel = 1.0e-6; /* Set desired relative error */
ans[1] = 1000.0; /* Set scaling reference */
ans[2] = 1000.0;
ans[3] = 1000.0;
```

<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>49.50</td>
<td>7.083</td>
<td>35.063</td>
</tr>
<tr>
<td></td>
<td>49.75</td>
<td>6.909</td>
<td>34.370</td>
</tr>
<tr>
<td></td>
<td>50.00</td>
<td>6.738</td>
<td>33.690</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 11.14. 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

---

70As illustrated, three of the arguments to `rkqs.c` must be flagged with a preceding `&` to be properly recognized by the routine.
Finally, we execute a loop in which, for each pass, the derivatives at the start of the current interval are evaluated, \texttt{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;
}
close( fptr );
```

A full listing of this program, which is named \texttt{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 \texttt{rkqs}, we obtain the actual solution and store it in the file \texttt{decayrkqs.dat} by

1. Creating the driving program \texttt{decayrkqs.c} and the subroutine \texttt{der} using an available text editor (or copying the files from the directory \texttt{$HEAD/cc}$).

2. Copying any necessary recipes to the default directory. In the present case, the header files \texttt{nr.h} and \texttt{nrutil.h} and the utility subroutine \texttt{nrutil.c} presumably remain from the first example. The recipes \texttt{rkqs.c} and \texttt{rkck.c} (whose importance will be proclaimed if the next step is attempted without it) must also be copied from the directory \texttt{$NRHEAD/recipes_c-ansi/recipes$}.

3. Compiling and linking the executable module with the statement

   ```c
   cc -o decayrkqs.xc decayrkqs.c rkqs.c rkck.c nrutil.c -lm
   ```

4. Running the program with the statement \texttt{./decayrkqs.xc}.

The first and last few lines in the file \texttt{decayrkqs.dat} 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>1000.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>0.25</td>
<td>975.310</td>
<td>24.383</td>
<td>0.307</td>
</tr>
<tr>
<td>1.35</td>
<td>873.693</td>
<td>117.971</td>
<td>8.336</td>
</tr>
<tr>
<td>3.49</td>
<td>705.048</td>
<td>246.407</td>
<td>48.545</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</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

\[
\begin{array}{cccc}
\text{t} & \text{A} & \text{B} & \text{C} \\
0.00 & 1000.000 & 0.000 & 0.000 \\
0.25 & 975.310 & 47.561 & 1.209 \\
0.50 & 927.743 & 69.581 & 2.676 \\
... & ... & ... & ... \\
49.75 & 6.909 & 34.370 & 958.722 \\
50.00 & 6.738 & 33.690 & 959.572 \\
50.25 & 6.572 & 33.022 & 960.406 \\
\end{array}
\]

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 \( \text{rel} (= 10^{-6}) \), we should be able to accept these values as accurate to \( \pm 0.001 \).

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

\[
x(0) = x_0 ; \quad \frac{dx}{dt}(0) = v_0
\]

and then determine the particular initial values that might be imposed so that the transient part of the solution is wiped out from the beginning, i.e., so that the motion is identically the steady-state motion from the moment the oscillator is set into motion. Assume that the oscillator is underdamped. (Note that, so that \( m \) will ultimately appear only in conjunction with \( b \) and \( k \), we have chosen to define \( f \) as a force per unit mass rather than simply a force.)

11.2. Take away the walls and the two springs connecting the blocks to the walls in the system of Fig. 11.5, let the masses be different \((m_1 \text{ and } m_2, \text{ say})\) and denote the constant of the one spring by \( k \). Suppose the blocks are constrained to move along a straight line. Measuring from an arbitrarily selected origin on that line, let the coordinates of the particles be \( x_1 \) and \( x_2 \), respectively. The equations of motion for this system are

\[
m_1 \frac{d^2x_1}{dt^2} = k(x_2 - x_1) ; \quad m_2 \frac{d^2x_2}{dt^2} = -k(x_2 - x_1)
\]

Let the system be put into motion with arbitrary initial conditions

\[
x_1(0) = x_{10} ; \quad x_2(0) = x_{20} ; \quad \frac{dx_1}{dt}(0) = v_{10} ; \quad \frac{dx_2}{dt}(0) = v_{20}
\]

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

$$\frac{As + B s^3}{\alpha s^4 + \beta s^2 + \gamma} = s \left( \frac{A + B s^2}{\alpha s^4 + \beta s^2 + \gamma} \right)$$

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

$$s \left( \frac{A + B s^2}{\alpha s^4 + \beta s^2 + \gamma} \right) = s \left( \frac{R}{s^2 + R'} + \frac{Q}{s^2 + Q'} \right) = \frac{Rs}{s^2 + R'} + \frac{Qs}{s^2 + Q'}$$

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

$$m \frac{d^2 x_1}{dt^2} = -k x_1 + k (x_2 - x_1)$$

$$m \frac{d^2 x_2}{dt^2} = -k (x_2 - x_1) + k (x_3 - x_2)$$

$$m \frac{d^2 x_3}{dt^2} = -k (x_3 - x_2) - k x_3$$

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

\[x(0) = x_0; \quad \frac{dx}{dt}(0) = 0\]

by explicitly taking the Laplace transform of the equation, solving that result for the Laplace transform of the solution, and then inverting that transform to obtain the solution itself.

11.6. Among the simplest of differential equations is the equation

\[
\frac{d^2 x}{dt^2} + \omega^2 x = 0
\]

that describes a simple harmonic oscillator. Generate a series solution to this equation and then verify that the solution thus generated agrees with the known solution

\[x(t) = A \cos \omega t + B \sin \omega t\]

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

\[
m \frac{d^2 r}{dt^2} = \frac{qQ}{4\pi\epsilon_0 r^3}
\]
Here, the force center is assumed to be at the origin, \( r \) is the position vector of the particle, and the equation is written in mksa units. Write this equation of motion out in Cartesian coordinates, cast it in dimensionless form, and determine the correspondences that must be adopted to convert the equations for an orbit in an electrostatic field into those for the orbit in a gravitational field, i.e., into Eq. (11.46).

11.10. Recast the system of six first-order equations in the last paragraph of Section 11.1.1 in dimensionless form, both when the viscous term is linear in the velocity and when it is quadratic in the velocity. Appropriate units for this recasting can be inferred from the discussion earlier in that section.

11.11. Translate the entire discussion—both dimensional and dimensionless—of the planetary problem in Section 11.1.7 from Cartesian coordinates \((x, y)\) to polar coordinates \((r, \phi)\), where \(x = r \cos \phi\) and \(y = r \sin \phi\). That is

(a) Show that the polar components of the equations of motion in the first instance are

\[
m \left[ \frac{d^2 r}{dt^2} - r \left( \frac{d \phi}{dt} \right)^2 \right] = f(r) ; \quad m \left( r \frac{d^2 \phi}{dt^2} + 2 \frac{d \phi}{dt} \frac{dr}{dt} \right) = 0
\]

(b) Recognizing that

\[
d \frac{dt}{dt} \left( m r^2 \frac{d \phi}{dt} \right) = m \left( r^2 \frac{d^2 \phi}{dt^2} + 2r \frac{dr}{dt} \frac{d \phi}{dt} \right)
\]

show that

\[m r^2 \frac{d \phi}{dt} = \text{constant} = L \implies \frac{d \phi}{dt} = \frac{L}{mr^2}\]

and then that

\[m \frac{d^2 r}{dt^2} = f(r) + \frac{L^2}{m r^3}\]

Here, \(L = m(x_0 v_{x0} - y_0 v_{y0})\) 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 \(d \phi/dt = \frac{L}{mr^2}\)—see Chapter 13—to find \(\phi(t)\).

(c) Translate the initial conditions to polar coordinates, finding that

\[r(0) = \sqrt{x_0^2 + y_0^2} \quad \frac{dr}{dt}(0) = \frac{x_0 v_{x0} + y_0 v_{y0}}{\sqrt{x_0^2 + y_0^2}}\]

\[\phi(0) = \text{atan} \frac{y_0}{x_0} \quad \frac{d \phi}{dt}(0) = \frac{x_0 v_{y0} - y_0 v_{x0}}{x_0^2 + y_0^2}\]

(d) Restrict the force to the inverse square gravitational force, finding that

\[\frac{d^2 r}{dt^2} = -\frac{GM}{r^2} + \frac{L^2}{m r^3}\]

(e) Recast the differential equations and initial conditions in dimensionless form, finding that

\[\frac{d^2 r}{dt^2} = -\frac{1}{r^2} + \frac{\beta}{r^3} ; \quad \frac{d \phi}{dt} = \pm \sqrt{\frac{\beta}{r^2}}\]

\[r(0) = \frac{r_0}{\ell} = \frac{\sqrt{x_0^2 + y_0^2}}{\ell} \quad ; \quad \phi(0) = \text{atan} \frac{y_0}{x_0}\]

\[\frac{dr}{dt}(0) = \frac{x_0 v_{x0} + y_0 v_{y0}}{\sqrt{x_0^2 + y_0^2} \sqrt{GM/\ell}} \quad ; \quad \frac{d \phi}{dt} = \frac{x_0 v_{y0} - y_0 v_{x0}}{(x_0^2 + y_0^2) \sqrt{GM/\ell^3}}\]

where \(\beta = L^2/(GM m^2 \ell)\) 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.
CHAPTER 11. SOLVING ORDINARY DIFFERENTIAL EQUATIONS

(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

\[
\frac{d^2y}{dx^2} + k^2 y = 0 \quad ; \quad y(0) = y(1) = 0
\]

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

\[
\frac{d^2y}{dx^2} \bigg|_{x=x_i} \approx \frac{y_{i+1} - 2y_i + y_{i-1}}{\Delta x^2} \quad ; \quad i = 1, 2, 3, \ldots, n - 1
\]

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.

11.14. Some radioisotopes exhibit a branched decay sequence such as

\[
\begin{array}{ccc}
A & k_1 & B \\
& k_2 & C \\
& k_3 & D \\
k_4 & & \\
\end{array}
\]

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 \( \bar{x}(0) = 1 \) and \( \bar{v}(0) = 0 \), produce graphs of \( \bar{x} \) versus \( \bar{t} \) for several values of \( \bar{\beta} \) on the interval \( 0.0 < \bar{\beta} < 2.0 \). Then using the knowledge that \( \bar{\beta} = b/m\omega = b/\sqrt{mk} \), and that \( \bar{t} = \omega t \), infer from your graphs the effect of changing \( m \) or \( k \) on the physical motion.

11.16. Explore the unforced, damped harmonic oscillator of Eq. (11.27) with \( F = 0 \) for the cases of critical damping (\( \bar{\beta} = 2.0 \)) and overdamping (\( \bar{\beta} > 2.0 \)).

11.17. Explore the behavior of the Van der Pol oscillator described in dimensionless form by the equation

\[
\frac{d^2\bar{x}}{dt^2} = \frac{d\bar{x}}{dt} (1 - \bar{x}^2) - \bar{x}
\]

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 \( t \), \( \theta / \sqrt{t} \) versus \( t \), and \( \theta / \sqrt{t} \) versus \( \theta \) (the phase plot). Obtain also a graph of period versus amplitude (initial displacement). Write several paragraphs describing your set up of the problem and presenting evidence for your discoveries. Optional: Try starting the pendulum at the bottom (0 initial angle) with several initial angular velocities. How large can the angular velocity be before the pendulum begins to swing over the top? Suggestion: Begin by introducing the dimensionless time \( t' = \sqrt{g/l} t \) so that the equation becomes \( \frac{d^2 \theta}{dt'^2} + \sin \theta = 0 \).

11.19. Suppose that the "gravitational" force were not inverse square but instead depended on some other \( b \) power of the radial coordinate. The dimensionless equations of motion then would be

\[
\frac{d^2 \pi}{d^2 t} = -\frac{\pi}{(\pi^2 + \pi^2)^b}; \quad \frac{d^2 \varphi}{d^2 t} = -\frac{\varphi}{(\varphi^2 + \varphi^2)^b}
\]

[Compare Eq. (11.46).] Of course, the equations reduce to those for the inverse square force if we simply set \( b = 3/2 \). For the planetary problem, find conditions that will generate a distinctly elliptical orbit for an attractive inverse square force \((b = 1.5)\). Then explore the effect on that orbit of distorting the force by changing the exponent in the denominator of the equations making \( b = 1.45, b = 1.35 \), 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 loops like a figure-eight around the two suns and/or you might see if your approach predicts what you would expect intuitively if you start the spaceship from rest at a point on the perpendicular bisector of the line joining the two suns. Make sure your solutions are generated to an adequate accuracy.

11.22. Suppose a particle of charge \( q \) and mass \( m \) is injected into a region of space containing constant, crossed electric and magnetic fields \( \mathbf{E} = E_x \hat{i} \) and \( \mathbf{B} = B_y \hat{k} \). 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

\[
m \frac{d^2 x}{dt^2} = q E_x + q B_y \frac{dy}{dt}; \quad m \frac{d^2 y}{dt^2} = -q B_x \frac{dx}{dt}; \quad m \frac{d^2 z}{dt^2} = 0
\]

for the specific fields of this exercise, express them in dimensionless form (note that \( \omega = q B_z / m \) is a frequency and \( E_x / B_z \) is a velocity), and thoroughly explore the behavior of the particle in this situation. Try to understand the motion intuitively. Hint: You should find that, in terms of an arbitrarily selected unit of length \( \ell \), the equations involve a single parameter \( q E_x / (m \omega^2 \ell) \), which can alternatively be written as \((E_x / B_z) / (\omega \ell)\) — the ratio of the velocity \( E_x / B_z \) determined by the fields to the characteristic velocity implied by your choice of a length unit and the frequency \( \omega \). Note that this exercise actually has more than one parameter, since the initial components of the velocity—probably expressed in units of \( \omega \ell \)—also influence the solution.

11.23. A particle having mass \( m \) and carrying charge \( q \) moves in the \( xy \)-plane while experiencing an electric field given by \( \mathbf{E}(x, y) = -\alpha y \hat{j} \), where \( \alpha \) is a constant. Assume that \( \alpha \) and \( q \) are both positive.
With \( \mathbf{r} = x \mathbf{i} + y \mathbf{j} \), the vector equation of motion for this particle then is
\[
m\frac{d^2 \mathbf{r}}{dt^2} = -qy \mathbf{j}
\]

(a) Show that, in component form, the equations of motion for this particle are
\[
\frac{d^2 x}{dt^2} = 0; \quad \frac{d^2 y}{dt^2} = -\frac{c}{m} y = -by
\]

(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
\[
\frac{dx}{dt} = a(y-x);
\]
\[
\frac{dy}{dt} = -xz + bx - y;
\]
\[
\frac{dz}{dt} = xy - \frac{c}{3} z
\]
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, while the true beauty of the trajectory is best seen using a three-dimensional space curve. Be sure to examine the path from several different vantage points in the space around the path, an objective most easily accomplished if the display of the path allows rotation of the path on the screen.

11.25. Recast the program 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 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

$$\frac{dx}{dt} = -k_1 x + k_2 xy ; \quad \frac{dy}{dt} = k_3 y - k_4 xy$$

where $k_1$ and $k_3$ are parameters describing the way each population would evolve in the absence of the other and $k_2$ and $k_4$ are parameters describing strength of the interaction between the two species, which we take to be proportional to the likelihood of an encounter between a member of one species and a member of the other species. Depending on the parameters and the initial populations $x(0) = x_0$, $y(0) = y_0$, the system may approach a stable equilibrium or, alternatively, one or the other of the populations may become extinct. Explore this system to determine conditions under which each of these circumstances occurs and write a paragraph or two describing your findings. Make sure your results are generated to an accuracy adequate to support your conclusions.

11.29. Examine the undamped, unforced harmonic oscillator carefully, using at least two different methods and several time steps. Monitor the accuracy of your solution by monitoring the constancy of the energy of the oscillator given (in dimensional form) by $E = \frac{1}{2}mv^2 + \frac{1}{2}kx^2$.

11.16.3 ... 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
11.D Listing of decay.c

/* PROGRAM decay.c */

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

main()
{

/***** Declare variables *****/

FILE *fptr; /* For file pointer */
float kA, kB; /* For parameters */
float t, A, B, C; /* For independent, dependent variables */
float dAdt, dBdt, dCdt; /* For derivatives */
float dt; /* For time increment */
int i, nsteps; /* Loop index, limit */

/***** Get parameters, time step, initial values *****/

printf( "Decay constant for A: " ); scanf( "%f", &kA );
printf( "Decay constant for B: " ); scanf( "%f", &kB );
printf( "Number of steps : " ); scanf( "%d", &nsteps );
printf( "Time step : " ); scanf( "%f", &dt );
printf( "Initial A : " ); scanf( "%f", &A );
printf( "Initial B : " ); scanf( "%f", &B );
printf( "Initial C : " ); scanf( "%f", &C );
t = 0.0;

/***** Open file, write label and initial values *****/

fptr = fopen( "decay_c.dat", "w" );
fprintf( fptr, " t A B C\n" );
fprintf( fptr, "%6.2f %10.3f %10.3f %10.3f\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;
printf( fptr, "%6.2f %10.3f %10.3f %10.3f\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
" );
fprintf( fptr, "%6.2f  %10.3f  %10.3f  %10.3f\n", t, n[1], n[2], n[3] );

for( i=1; i<=200; i++ )
{
    der( t, n, dndt );
rk4( n, dndt, 3, t, dt, n, der);
t = t + dt;
    fprintf( fptr, "%6.2f  %10.3f  %10.3f  %10.3f\n", t, n[1], n[2], n[3] );
}
fclose( fptr );
}
11.F Listing of decayrkqs.c

/* PROGRAM decayrkqs.c */

#include <stdio.h>
#include <math.h>
#include "nr.h"
#include "nrutil.h"

float kA, kB; /* For parameters */

void der( float t, float n[], float dndt[] )
{
    dndt[1] = -kA*n[1]; /* Calculate dA/dt */
    dndt[3] = kB*n[2]; /* Calculate dC/dt */
}

main()
{
    FILE *fptr; /* For file pointer */
    float n[3+1], dndt[3+1]; /* For variables and derivatives */
    float rel, ans[3+1]; /* For error control */
    float t, dttry, dtnext, dtact; /* For time, time steps */
    kA = 0.1; kB = 0.1; /* Set parameters */
    dttry = 0.25; /* Set time step */
    n[1] = 1000.0; n[2] = 0.0; /* Initialize dependent variables */
    n[3] = 0.0;
    t = 0.0; /* Set initial time */
    rel = 1.0e-6; /* Set desired relative error */
    ans[1] = 1000.0; /* Set scaling reference */
    ans[2] = 1000.0;
    ans[3] = 1000.0;

    fptr = fopen( "decayrkqs_c.dat", "w" );
    fprintf( fptr, " t A B C\n" );
    fprintf( fptr, "%6.2f %10.3f %10.3f %10.3f\n", t, n[1], n[2], n[3] );

    while ( t <= 50.0 )
    {
        der( t, n, dndt );
        rkqs(n, dndt, 3, &t, dttry, rel, ans, &dtact, &dtnext, der );
        fprintf( fptr, "%6.2f %10.3f %10.3f %10.3f\n", t, n[1], n[2], n[3] );
        dttry = dtnext;
    }
    fclose( fptr );
}
Chapter 13

Evaluating Integrals

Frequently the answer to an interesting question in physics is given by—or can be cast in the form of—an integral, often as a function of the upper limit or as a parameter in the integrand. Sometimes, that integral can be evaluated in closed form. More often, however, the integral is analytically intractable and must be approached numerically. We begin this chapter by identifying several physical situations, the full addressing of which requires evaluating an integral, perhaps as a function of one or more parameters. Then we illustrate how to use symbolic algebra systems to approach those that can be evaluated analytically, describe a few of many available numerical algorithms (with attention to their accuracy), and—finally—describe ways to evaluate representative integrals using a variety of numerical approaches and computational tools. When parameters are involved, we also illustrate how to plot graphs of the integrals as functions of those parameters.

We shall classify each integral in one of three categories, since the approach to its numerical evaluation will depend on this classification. Integrals in the first category will simply be a number; their numerical evaluation involves a single invocation of one or another basic algorithm. In more complicated—and more interesting—cases, the integral will be a function of a parameter, which may appear in the limits of integration (second category) or embedded in the integrand (third category); the numerical evaluation of these integrals as a function of the parameter will involve repeated invocation of one or another basic algorithm within a loop.

13.1 Sample Problems

In this section, we identify several physical contexts in which integrals arise, and we determine a representative integral for each case.

13.1.1 One-Dimensional Trajectories

The motion of a particle of mass $m$ in one dimension under the action of a force $f$ satisfies Newton’s second law, equivalent to the two equations

$$\frac{dp}{dt} = f \quad \text{and} \quad \frac{dx}{dt} = v$$

which are to be solved subject to the prescribed initial conditions $x(0) = x_0$, $v(0) = v_0$, and $p(0) = p_0$. Here, $x$, $v$, and $p$, are the position, velocity, and momentum of the particle, respectively, and, with $c$ standing for the speed of light, the relationship between $p$ and $v$ assumes one of the forms

$$p = \frac{mv}{\sqrt{1 - v^2/c^2}} \quad \text{or} \quad p = mv$$

(13.2)
depending on whether the motion is relativistic or nonrelativistic. If the force happens to depend only on \( t \), the solution to these two differential equations can be expressed as the explicit integrals

\[
p(t) = p_0 + \int_0^t f(t') \, dt' \quad \text{and} \quad x(t) = x_0 + \int_0^t v(t') \, dt'
\]  

(13.3)

and the physical problem of predicting the trajectory reduces to the mathematical problem of evaluating two integrals, finding the momentum from the first integral in Eq. (13.3), then solving for the velocity \( v(t) \) using the appropriate member of Eq. (13.2), and finally finding \( x(t) \) from the second integral in Eq. (13.3). Each integral is a function of at least one parameter—the upper limit \( t \)—and will also depend on additional quantities (e.g., \( m, c, \ldots \)) in the integrand (unless a dimensionless casting happens to suppress them). These integrals fall into either the second or the third of our three categories.

If, on the other hand, the force happens to depend only on \( x \), we can recast the computational task by noting first that

\[
\frac{d^2 x}{dt^2} = \frac{dv}{dt} = \frac{dv}{dx} \frac{dx}{dt} = v \frac{dv}{dx}
\]  

(13.4)

Then, Newton’s second law becomes

\[
\frac{m}{d^2 x}{dt^2} = f(x) \quad \Rightarrow \quad mv \frac{dv}{dx} = f(x) \quad \Rightarrow \quad mv \, dv = f(x) \, dx
\]  

(13.5)

Finally, by integrating this last expression from initial values to general values at some other time, we find that

\[
m \int_{v_0}^v v' \, dv' = \frac{1}{2} mv^2 - \frac{1}{2} mv_0^2 = \int_{x_0}^x f(x') \, dx'
\]  

(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 \( \mathbf{r}_{\text{cm}} \) of the center of mass of an object is given by

\[
\mathbf{r}_{\text{cm}} = \frac{1}{M} \int \mathbf{r} \, dm
\]  

(13.8)

where \( \mathbf{r} \) locates a representative element of the object, \( dm \) is the mass of that element, \( M = \int dm \) is the total mass of the object, and the integral extends over the region of space (volume, surface, or line) occupied by the object. More specifically, if the object lies in a plane and polar coordinates \((r, \phi)\) are appropriate, we would write this integral more explicitly in the form

\[
\mathbf{r}_{\text{cm}} = \frac{1}{M} \int \sigma(r, \phi) \, r \, dr \, d\phi = \frac{1}{M} \int \sigma(r, \phi) \left( r \cos \phi \, \mathbf{i} + r \sin \phi \, \mathbf{j} \right) r \, dr \, d\phi
\]  

(13.9)

where \( \sigma(r, \phi) \) is the mass per unit area of the object and \( r \, dr \, d\phi \) is the area of the chosen element.
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

\[
\mathbf{r}_{\text{cm}} = \frac{1}{M} \int_{0}^{a} \int_{0}^{\pi} \frac{M}{2 \pi a^2} \left( r \cos \phi \, \hat{i} + r \sin \phi \, \hat{j} \right) r \, dr \, d\phi = \frac{2}{\pi a^2} \int_{0}^{a} \int_{0}^{\pi} r^2 \left( \cos \phi \, \hat{i} + \sin \phi \, \hat{j} \right) \, dr \, d\phi
\]

(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

\[
\frac{\mathbf{r}_{\text{cm}}}{a} = \frac{2}{\pi} \int_{0}^{1} \int_{0}^{\pi} \lambda^2 \left( \cos \phi \, \hat{i} + \sin \phi \, \hat{j} \right) d\phi \, d\lambda
\]

(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)
CHAPTER 13. EVALUATING INTEGRALS

Figure 13.2: A simple pendulum.

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

\[
dt = \pm \sqrt{\frac{l}{2g}} \frac{d\theta}{\sqrt{\cos \theta - \cos \theta_0}} \quad \Rightarrow \quad \int_0^{T/4} dt = \frac{T}{4} = \sqrt{\frac{l}{2g}} \int_{\theta_0}^{\theta_0} \frac{d\theta}{\sqrt{\cos \theta - \cos \theta_0}}
\]
13.1. SAMPLE PROBLEMS

where we have integrated over one-quarter of the period \( 0 < t < T/4 \) in time and over one-quarter of the swing \( 0 < \theta < \theta_0 \) in angle, and we have taken the positive sign because, in its motion over this interval, the pendulum indeed has positive angular velocity. Further, we have assumed from symmetry that the full period \( T \) is four times the time required for the pendulum to swing from its lowest point to its highest point.

This integral can be recast in numerous ways. Anticipating an existing standard function, we invoke the half angle identity \( \cos \theta = 1 - 2\sin^2(\theta/2) \), finding that

\[
T = 2\sqrt{\frac{l}{g}} \int_0^{\theta_0} \frac{d\theta}{\sqrt{\sin^2(\theta_0/2) - \sin^2(\theta/2)}}
\]

We then introduce the variable \( \phi \) defined by \( \sin(\theta/2) = \sin(\theta_0/2) \sin \phi \) to find that

\[
T(k) = 4\sqrt{\frac{l}{g}} \int_0^{\pi/2} d\phi \sqrt{1 - k^2 \sin^2 \phi}
\]

where the (dimensionless) parameter \( k = \sin(\theta_0/2) \) is determined by the amplitude of the pendulum’s motion. By writing \( T(k) \), we have drawn attention in the notation to the dependence of the period on \( k \) (and hence on the amplitude). Note also that, when the amplitude is small, \( k \approx 0 \) and the integral can be readily evaluated to yield that \( T_0 = 2\pi\sqrt{l/g} \), which—reassuringly—agrees with the known value quoted at the beginning of this subsection. Then, expressing the period \( T(k) \) in units of \( T_0 \), we find finally that

\[
\frac{T(k)}{T_0} = \frac{2}{\pi} \int_0^{\pi/2} \frac{d\phi}{\sqrt{1 - k^2 \sin^2 \phi}} = \frac{2}{\pi} K(k)
\]

Here, for purposes of notation alone, we have introduced the integral

\[
K(k) = \int_0^{\pi/2} \frac{d\phi}{\sqrt{1 - k^2 \sin^2 \phi}}
\]

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

\[
G(x) = \frac{1}{\sqrt{2\pi}\sigma^2} e^{-(x-\mu)^2/(2\sigma^2)}
\]

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

\[
P(\mu - \alpha \sigma < x < \mu + \alpha \sigma) = \int_{\mu-\alpha \sigma}^{\mu+\alpha \sigma} G(x) \, dx = \frac{2}{\sqrt{\pi}} \int_0^{\alpha/\sqrt{2}} e^{-s^2} \, ds = \text{erf} \left( \frac{\alpha}{\sqrt{2}} \right)
\]

where \( s = (x - \mu)/(\sigma \sqrt{2}) \) and, for purposes of notation alone, we have introduced the integral

\[
\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-s^2} \, ds
\]
which defines a standard, tabulated function known as the \textit{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.

\subsection*{13.1.6 The Cornu Spiral}

The two integrals

\begin{equation}
C(u) = \int_0^u \cos \left( \frac{\pi^2 t^2}{2} \right) \, dt \quad ; \quad S(u) = \int_0^u \sin \left( \frac{\pi^2 t^2}{2} \right) \, dt \tag{13.27}
\end{equation}

which are called the \textit{Fresnel integrals}, appear in the study of Fresnel diffraction. Together, they define the \textit{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.

\subsection*{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 \( dq' \) located at \( r' \) and, in mks units, we find the electric field \( E(r) \) and the electrostatic potential \( V(r) \) at the point \( r \) by evaluating the integrals

\begin{equation}
E(r) = \frac{1}{4\pi \varepsilon_0} \int \frac{r - r'}{|r - r'|^3} \, dq' \quad \text{and} \quad V(r) = \frac{1}{4\pi} \int \frac{dq'}{|r - r'|} \tag{13.28}
\end{equation}

which extend over all charges in the source. Similarly, for a source consisting of a steady current \( I' \) in a wire, we identify an element \( dr' \) of the wire located at \( r' \) and, in mks units, find the magnetic field \( B(r) \) and the magnetic vector potential \( A(r) \) at the point \( r \) by evaluating the integrals

\begin{equation}
B(r) = \frac{\mu_0}{4\pi} \int \frac{I' \, dr' \times (r - r')}{|r - r'|^3} \quad \text{and} \quad A(r) = \frac{\mu_0}{4\pi} \int \frac{I' \, dr'}{|r - r'|} \tag{13.29}
\end{equation}

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,

\begin{equation}
r - r' = (a \cos \phi - a \cos \phi') \hat{i} + (a \sin \phi - a \sin \phi') \hat{j} + z \hat{k} \tag{13.30}
\end{equation}

so

\begin{align*}
|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} \tag{13.31}
\end{align*}

Placing these results into the second member of Eq. (13.28) and recognizing that \( dq' = \lambda a \, d\phi' \), we at last find that

\begin{equation}
V_{\text{one}}(r, \phi, z) = \frac{1}{4\pi \varepsilon_0} \int_0^{2\pi} \frac{\lambda a \, d\phi'}{|r^2 + a^2 - 2ar \cos(\phi' - \phi) + z^2|^{1/2}} \tag{13.32}
\end{equation}
Figure 13.3: The potential produced by two charged rings. Part (a) shows the geometry of ultimate interest. Part (b) shows the simple situation with one ring used as a stepping stone.

The entire integral assumes a simpler appearance, however, on the variable \( \alpha = \phi' - \phi \), becoming

\[
V_{\text{one}}(r, \phi, z) = \frac{\lambda a}{4\pi \varepsilon_0} \int_0^{2\pi} d\alpha \frac{1}{ [r^2 + a^2 - 2ar \cos \alpha + z^2]^{1/2} } \tag{13.33}
\]

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

\[
V_{\text{midplane}}(r) = V_{\text{one}}(r, \phi, -a) + V_{\text{one}}(r, \phi, a) = \frac{2\lambda a}{4\pi \varepsilon_0} \int_0^{2\pi} d\alpha \frac{1}{[r^2 + 2a^2 - 2ar \cos \alpha + 2]^{1/2} } \tag{13.34}
\]

Finally, introducing the variable \( s \) defined by \( r = sa \) to express the radial coordinate in dimensionless terms, we find the expression

\[
V_{\text{two}}(s) = V_{\text{midplane}}(sa) = \frac{2\lambda}{4\pi \varepsilon_0 a} \int_0^{2\pi} d\alpha \frac{1}{[s^2 + 2s \cos \alpha + 2]^{1/2} } \tag{13.35}
\]

or, even better, the expression

\[
V(s) = \frac{\sqrt{2}}{2\pi} \int_0^{2\pi} d\alpha \frac{1}{[s^2 - 2s \cos \alpha + 2]^{1/2} } \tag{13.36}
\]

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

\[
P(x_1 < x < x_2) = \int_{x_1}^{x_2} |\psi(x)|^2 \, dx \tag{13.37}
\]

gives the probability of finding the quantum system with its coordinate somewhere between \( x = x_1 \) and \( x = x_2 \).

More specifically, we remember that the wave function for a quantum harmonic oscillator in its ground state is given by

\[
\psi(x) = \left( \frac{m\omega}{\pi\hbar} \right)^{1/4} e^{-y^2/2} \tag{13.38}
\]

where \( \hbar \) is Planck’s constant, \( m \) is the mass of the oscillator and, with \( k \) the spring constant and \( a = \sqrt{\hbar\omega/k} \), \( \omega = \sqrt{k/m} \) is the frequency of the oscillator; \( y = x/a \) is a dimensionless coordinate.

Since the classical turning point of the oscillator occurs when its energy (\( \frac{1}{2}\hbar\omega \) for the ground state) is equal to the potential energy \( \frac{1}{2}kx_{\text{turn}}^2 \), the turning point of this oscillator is given by

\[
\frac{1}{2}\hbar\omega = \frac{1}{2}kx_{\text{turn}}^2 \implies x_{\text{turn}} = \sqrt{\frac{\hbar\omega}{k}} = a \tag{13.39}
\]

which provides a classical interpretation for the parameter \( a \). The probability that the particle in the ground state of a quantum harmonic oscillator will be found in the classically forbidden region (i.e., somewhere outside the classical turning point) is given by the integral

\[
P\left(|x| > |x_{\text{turn}}|\right) = \int_{-\infty}^{-a} |\psi(x)|^2 \, dx + \int_{a}^{\infty} |\psi(x)|^2 \, dx = 1 - \int_{-a}^{a} |\psi(x)|^2 \, dx \tag{13.40}
\]

Finally, after substituting the wave function and recasting the entire integral as an integral on the variable \( y \), we find that

\[
P\left(|x| > |x_{\text{turn}}|\right) = 1 - \frac{1}{\sqrt{\pi}} \int_{-1}^{1} e^{-y^2} \, dy = 1 - \frac{2}{\sqrt{\pi}} \int_{0}^{1} e^{-y^2} \, dy = 1 - \text{erf}(1) \tag{13.41}
\]

which contains no parameters and thus falls into the first of our three categories.

13.1.9 Expansion in Orthogonal Functions

Suppose we have identified a set of functions \( \phi_i(x) \), \( i = 1, 2, 3, \ldots \), that have the property

\[
\int_{a}^{b} \phi_i(x) \phi_j(x) \, w(x) \, dx = N_j \delta_{ij} \tag{13.42}
\]

where \( N_j \) is a constant, \( \delta_{ij} \) is the Kronecker delta (which has the value 1 when the indices are equal and the value 0 otherwise), \( w(x) \) is a known weight function, and \( a \) and \( b \) define a known interval. These functions are said to be orthogonal with weight \( w(x) \) on the interval \( a \leq x \leq b \). The members of this set provide a basis in terms of which any arbitrary function \( f(x) \) defined on the same interval can be expanded in the form

\[
f(x) = \sum_{n} a_n \phi_n(x) \tag{13.43}
\]
13.4 Evaluating Integrals Symbolically with Mathematica

Though the argument we will here present is not mathematically rigorous, the expansion coefficients $a_n$ can be quickly determined by multiplying Eq. (13.43) by $w(x) \phi_j(x)$, integrating over the interval $a \leq x \leq b$, and exchanging the order of integration and summation to find that

$$\int_a^b w(x) \phi_j(x) f(x) \, dx = \sum_n a_n \int_a^b w(x) \phi_j(x) \phi_n(x) \, dx = \sum_n a_n N_n \delta_{nj} = N_j a_j \quad (13.44)$$

We conclude that, if we know either of $f(x)$ or $a_n$, we can determine the other, i.e., that

$$f(x) = \sum_n a_n \phi_n(x) \quad \iff \quad a_n = \frac{1}{N_n} \int_a^b w(x) \phi_n(x) f(x) \, dx \quad (13.45)$$

The determination of the coefficients in this expansion of a known function clearly involves the evaluation of integrals, which explains the inclusion of this example in this chapter.

While many sets of orthogonal functions could be enumerated (see exercises), probably the most common set is

$$\{1, \sin \frac{n\pi x}{l}, \cos \frac{m\pi x}{l}; \quad n, m = 1, 2, 3, \ldots\} \quad (13.46)$$

Direct evaluation of the integral of each member of this set with all other members will reveal that these functions are orthogonal on the interval $-l \leq x \leq l$ with weight $w(x) = 1$, i.e., that

$$\int_{-l}^l 1 \times 1 \, dx = 2l \quad ; \quad \int_{-l}^l 1 \times \sin \frac{n\pi x}{l} \, dx = 0 \quad ; \quad \int_{-l}^l 1 \times \cos \frac{n\pi x}{l} \, dx = 0 \quad (13.47)$$

$$\int_{-l}^l \sin \frac{n\pi x}{l} \sin \frac{m\pi x}{l} \, dx = l \delta_{nm} \quad ; \quad \int_{-l}^l \sin \frac{n\pi x}{l} \cos \frac{m\pi x}{l} \, dx = 0 \quad (13.48)$$

$$\int_{-l}^l \cos \frac{n\pi x}{l} \cos \frac{m\pi x}{l} \, dx = l \delta_{nm} \quad ; \quad \int_{-l}^l \cos \frac{n\pi x}{l} \sin \frac{m\pi x}{l} \, dx = 0 \quad (13.49)$$

The expansion

$$f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty} \left( a_n \cos \frac{n\pi x}{l} + b_n \sin \frac{n\pi x}{l} \right) \quad (13.50)$$

of a function $f(x)$ that is periodic with period $2l$ in this set of orthogonal functions is called a Fourier series. The coefficients are given by the integrals

$$a_n = \frac{1}{l} \int_{-l}^l f(x) \cos \frac{n\pi x}{l} \, dx \quad ; \quad b_n = \frac{1}{l} \int_{-l}^l f(x) \sin \frac{n\pi x}{l} \, dx \quad (13.51)$$

13.4 Evaluating Integrals Symbolically with Mathematica

The Mathematica command for symbolic integration is Integrate. The first argument in this command 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

Integrate[ Sin[k*x], x ]

evaluates the indefinite integral $\int \sin(kx) \, dx$ while the statement

Integrate[ Sin[k*x], {x, a, b} ]

evaluates the definite integral $\int_a^b \sin(kx) \, dx$.
Integrate[ Sin[k*x], {x, 0, Pi} ]

evaluates the definite integral \( \int_0^\pi \sin(kx) \, dx \). If Mathematica 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. Mathematica also evaluates improper integrals, recognizing the symbols \( \text{Infinity} \) for \( +\infty \) and \( -\text{Infinity} \) 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, Mathematica checks for these singularities automatically and prints a warning if evidence of a singularity is found.

In this section, we illustrate the use of the command \texttt{Integrate} to evaluate some of the integrals deduced in Section 13.1. Beyond integration \textit{per se}, we also show how, in many cases, Mathematica can be used to set up the integral as well. To abbreviate the presentation of Mathematica dialogs, we make liberal use of terminating semicolons to suppress intermediate output. You are therefore urged to duplicate the dialogs in an actual session with Mathematica, removing semicolons so that intermediate output is explicitly displayed.

13.4.1 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

\[
p(t) = \int_0^t f \, dt', \quad x(t) = \int_0^t v(t') \, dt', \quad \text{and} \quad p(t) = \frac{mv(t)}{\sqrt{1 - v(t)^2/c^2}}
\]  

(13.52)

The following “conversation” with Mathematica will find the desired quantities and explore a few of their properties:\footnote{Mathematica statements are shown on the left; comments describing the statements are shown on the right.}

\begin{verbatim}
In[1]:= p = Integrate[ f, {tp, 0, t} ]  \quad \text{(Integrate (constant) \( f \) to find \( p \).)}
Out[1]= ft

In[2]:= p == m * v / Sqrt[ 1 - v^2/c^2 ]  \quad \text{(Relate momentum to velocity.)}
Out[2]= ft == mv/\sqrt{1 - v^2/c^2}

In[3]:= soln = Solve[ %, v ]  \quad \text{(Extract solution as an expression. (Physically, \( v \) and \( f \) must have the same sign.))}
Out[3]= \{ \{ v \to -\frac{cft}{\sqrt{c^2m^2 + f^2t^2}} \}, \{ v \to \frac{cft}{\sqrt{c^2m^2 + f^2t^2}} \} \}

In[4]:= v = soln[[2,1,2]]
Out[4]= \frac{cft}{\sqrt{c^2m^2 + f^2t^2}}

In[5]:= Limit[ v, t \to \text{Infinity} ]  \quad \text{(Find limit of \( v \) as \( t \to \infty \).)}
Out[5]= \frac{cf}{\sqrt{f^2}}

In[6]:= Simplify[ %, f > 0 ]
Out[6]= c
\end{verbatim}
In[7]:= vp = v /. t -> tp;       Substitute tp for t in v.
In[8]:= x = Integrate[ vp, { tp, 0, t} , Assumptions->{c>0,m>0,t>0,f>0} ]
Out[8]= \[Integral] (-cm + \sqrt{c^2m^2 + f^2t^2})
f
In[9]:= Series[ v, { t, 0, 4 } ];       Find limit of v for small t.
In[10]:= vclassical = Simplify[ %, {c>0, m>0, f>0} ]
Out[10]= ft^3 - \frac{f^3}{2c^2m^3} + O[t]^6
In[11]:= Series[ x, { t, 0, 5 } ];       Find limit of x for small t.
In[12]:= xclassical = Simplify[ %, {c>0, m>0, f>0 } ]
Out[12]= \frac{ft^2}{2m} - \frac{f^3}{8c^2m^3} + O[t]^6
In[13]:= Quit[]

Reassuringly, the velocity approaches c as t → ∞ 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.

13.4.2 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

In[1]:= rr = {r*Cos[\[Phi]], 0};
Assign r.
In[2]:= \[Sigma] = m / (Pi * a^2 / 2);       Evaluate \sigma, which is constant.
In[3]:= integ = rr * \[Sigma] * r;        Evaluate integrand.
In[4]:= fstint = Integrate[ integ,
\{ \[Phi], 0, Pi \} ];                  Evaluate integral on \phi.
In[5]:= scndint = Integrate[ fstint,
\{ r, 0, a \} ];                      Evaluate integral on r.

In[6]:= rcm = scndint/m                  Multiply each component by 1/m.
Out[6]= \{ 0, -\frac{4a}{3\pi}, 0 \}
In[7]:= N[ % ]               Obtain floating evaluation.
Out[7]=\{ 0., -0.424413a, 0. \}
In[8]:= Quit[]

Since only the y component of r_{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, inci-
didentally, that Mathematica has at lines In[4] and In[5] effected an element-by-element integration
of the components of the list with a single statement.

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.
13.4.3 Moment of Inertia; Radius of Gyration

The evaluation by Mathematica of the moment of inertia of a semicircular plate as given by Eq. (13.16) is quick and straightforward. We need only the statements

\[
\text{In[1]}:= \text{Integrate}[\lambda^2\sqrt{1-\lambda^2}, \{\lambda, 0, 1\}]
\]

\[
\text{Out[1]}= \frac{\pi}{16}
\]

\[
\text{In[2]}:= II = m*a^2*(4/Pi)*%
\]

\[
\text{Out[2]}= \frac{a^2m}{4}
\]

\[
\text{In[3]}:= k = \text{Simplify}[\text{Sqrt}[II/m], a>0]
\]

\[
\text{Out[3]}= \frac{a}{2}
\]

\[
\text{In[4]}:= \text{Quit[]}
\]

to evaluate the necessary integral and find both the moment of inertia \(I = ma^2/4\) and the radius of gyration \(k = a/2\).

13.4.4 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, \(r\) and \(r'\) 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

\[
V(x,y,z) = \frac{\lambda}{4\pi\epsilon_0} \int_{-a}^{+a} \frac{dz'}{|r-r'|}
\]

(Eq. 13.53)

Electing cylindrical coordinates \((r, \phi, z)\), we set \(r' = z' \hat{k}\) and \(r = r \cos \phi \hat{i} + r \sin \phi \hat{j} + z \hat{k}\). Then we construct the integrand and evaluate the integral with the statements

\[
\text{In[1]}:= \text{Get}["$HEAD/mathematica/crossdot.m"]
\]

\[
\text{Define ludot and lucross.}
\]

\[
\text{In[2]}:= \text{rr} = \{r*\text{Cos}[\Phi], r*\text{Sin}[\Phi], z\};
\]

\[
\text{Assign r in cylindrical coordinates.}
\]

\[
\text{In[3]}:= \text{rp} = \{0, 0, zp\};
\]

\[
\text{Assign r'}.\]

\[
\text{In[4]}:= \text{sep} = \text{rr} - \text{rp};
\]

\[
\text{Evaluate r-r'.}
\]

\[
\text{In[5]}:= \text{ludot[sep, sep]};
\]

\[
\text{Evaluate |r-r'|^2.}
\]

\[
\text{In[6]}:= \text{Simplify[ % ];}
\]

\[
\text{Recognize cos^2 \phi + sin^2 \phi = 1.}
\]

\[
\text{In[7]}:= \text{den} = \text{Sqrt}[ % ];
\]

\[
\text{Bind |r-r'| to den.}
\]

\[
\text{In[8]}:= \text{cnst} = 1/(4*\text{Pi}*[\text{Epsilon}]0);
\]

\[
\text{Bind 1/4\pi\epsilon_0 to cnst.}
\]

\[
\text{In[9]}:= \text{integ} = \{\lambda*cnst\} / \text{den}
\]

\[
\text{Set integrand.}
\]

\[
\text{Out[9]}= \frac{\lambda}{4\pi\sqrt{r^2 + (z-z')^2}} \epsilon_0
\]

Then, we find \(V(r,z)\), with \(r = \sqrt{x^2 + y^2}\) by executing the statement

---

4 The variable \(I\) in Mathematica is reserved for \(\sqrt{-1}\), so we use \(II\) for the moment of inertia.

5 The file \$HEAD/mathematica/crossdot.m was described in Section 8.11 and exists in the public directory structure. (The symbol \$HEAD is defined in the Local Guide.) It defines the two functions \text{lucross[v1, v2]} and \text{ludot[v1, v2]} for evaluating the cross and dot products of two three-component vectors when those vectors are conveyed by lists.
13.4. EVALUATING INTEGRALS SYMBOLICALLY WITH MATHEMATICA

Figure 13.4: A line charge lying on the $z$ axis.

In[10]:= \( V = \text{Integrate} [ \text{integ}, \{ zp, -a, a \}, \text{Assumptions} \to \{ a>0, z>0, r>0 \} ] \)

\( \lambda \log \left[ \frac{a + z + \sqrt{r^2 + (a + z)^2}}{-a + z + \sqrt{r^2 + (-a + z)^2}} \right] \)

\( \frac{4\pi\varepsilon_0}{r} \)

While correct, this result is certainly not particularly transparent. Let us test it 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. For the first limit, we execute the statements

In[11]:= \( V_0 = V / . z \to 0 \)

\( \lambda \log \left[ \frac{a + \sqrt{r^2 + a^2}}{-a + \sqrt{r^2 + a^2}} \right] \)

\( \frac{4\pi\varepsilon_0}{r} \)

In[12]:= \( V_0 / . a \to s r \);

In[13]:= \( \text{Series} [ \%, \{ s, 0, 2 \} ] \);

In[14]:= \( \text{Simplify} [ \%, r>0 ] \)

\( \frac{\lambda s^2}{2\pi\varepsilon_0} + O[s]^3 \)

In[15]:= \( \text{Normal} [ \% ] \);

In[16]:= \( \% / . s \to a/r \);

In[17]:= \( \% / . \{ \lambda \to q/(2a) \} \)

\( \frac{q}{4\pi\varepsilon_0 r} \)

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 might think to invoke a statement like `Series[V0, \{r,0,2\}]`. That approach, however, is problematic because the denominator under the logarithm in \( V_0 \) is zero at \( r = 0 \) when \( a > 0 \). The expression \( V_0 \) does not have a very well-behaved series expansion about the point \( r = 0 \). As an alternative, we might notice that the numerator and denominator under the logarithm in \( V_0 \) are well-behaved in the vicinity of \( r = 0 \). In particular

\[
\text{In}[18]:= \text{Series}[-a + Sqrt[a^2 + r^2], \{r,0,2\}, \text{Assumptions}\to a>0];
\]
\[
\text{In}[19]:= T1 = \text{Normal}[\text{Simplify}[% , a>0]]
\]
\[
\text{Out}[19]= \frac{r^2}{2a}
\]

\[
\text{In}[20]:= \text{Series}[a + Sqrt[a^2 + r^2], \{r,0,2\}, \text{Assumptions}\to a>0];
\]
\[
\text{In}[21]:= T2 = \text{Normal}[\text{Simplify}[% , a>0]]
\]
\[
\text{Out}[21]= 2a + \frac{r^2}{2a}
\]

Substituting these expansions into \( V_0 \) with the statements

\[
\text{In}[22]:= \text{SmallR} = V_0 \/. \{-a+Sqrt[a^2+r^2]\to T1, a+Sqrt[a^2+r^2]\to T2\}
\]
\[
\text{Out}[22]= \frac{\lambda}{4\pi\epsilon_0} \ln \left[ \frac{2a(2a + r^2/2a)}{r^2} \right]
\]

we find a much simpler, though approximate expression. Further, we can now see why a well-behaved series expansion does not exist. As \( r \) becomes ever smaller, the denominator in this logarithm approaches zero and the logarithm diverges. The numerator, however, approaches \( 4a^2 \) as \( r \to 0 \). Thus, we can simplify the expression further by ignoring the term \( r^2/2a \) in the numerator. The statement

\[
\text{In}[23]:= \text{SmallR} = \text{SmallR} \/. \ 2a + r^2/(2a) \to 2a
\]
\[
\text{Out}[23]= \frac{\lambda}{4\pi\epsilon_0} \ln \left[ \frac{4a^2}{r^2} \right]
\]
\[
\text{In}[23]:= \text{Quit}[]
\]

effects this further simplification. Indeed, if we recognize the equivalent form,

\[
-\frac{\lambda}{2\pi\epsilon_0} \ln \frac{r}{a} + \frac{\lambda \ln 2}{2\pi\epsilon_0}
\]

we see that our result 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.

### 13.4.5 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 single loop. In general, the magnetic field \( B(r) \) is given by the Biot-Savart law, which is the first
13.4. EVALUATING INTEGRALS SYMBOLICALLY WITH MATHEMATICA 365

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\(^6\)

\begin{verbatim}
In[1]:= Get[ "$HEAD/mathematica/crossdot.m" ];
In[2]:= r = { 0, 0, s * a };           Assign r.
In[3]:= rp = {a*Cos[\[Phi]],a*Sin[\[Phi]],0}; Assign r'.
In[4]:= drp = {-a*Sin[\[Phi]],a*Cos[\[Phi]],0}; Assign dr'.
In[5]:= sep = r - rp;                    Evaluate \(r - r'\).
In[6]:= ludot[ sep, sep ];             Evaluate \(|r - r'|^2\).
In[7]:= mag2 = Simplify[ % ]; Recognize \(\cos^2 \phi + \sin^2 \phi = 1\); set
In[8]:= lucross[ drp, sep ];          \(\mathbf{r} - \mathbf{r}'\).
In[9]:= num = Simplify[ % ]; Recognize \(\cos^2 \phi + \sin^2 \phi = 1\).
In[10]:= integ = num*\[Mu]0*i/(4*Pi*mag2^(3/2)); Evaluate \(\mathbf{r} - \mathbf{r}'\).
In[11]:= integ = Simplify[ %, a>0 ]; Evaluate \(\mathbf{d}n\times (\mathbf{r} - \mathbf{r}')\).
Out[11]= \{i\[Mu]0 s \cos [\[Phi]], i\[Mu]0 s \sin [\[Phi]], i\[Mu]0 4 \pi (1 + s^2)^(3/2) \}
In[12]:= B1 = Integrate[ integ, { \[Phi], 0, 2*Pi } ] Integrate around the loop.
Out[12]= \{0, 0, \frac{i\[Mu]0}{2a(1 + s^2)^(3/2)} \}
In[13]:= B1z = B1[[3]]; Extract the \(z\) component.
In[14]:= B1c = B1z /. s -> 0; Find field at center.
In[15]:= B1z = B1z / B1c Normalize field of single loop.
Out[15]= \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

\begin{verbatim}
In[16]:= Bpair = (B1z /. s -> s-c) + (B1z /. s -> s+c); Find field of two loops, one at \(s = c\) and
In[17]:= Bpairc = Bpair /. s -> 0; the other at \(s = -c\).
In[18]:= Bpair = Bpair / Bpairc Normalize field of two loops.
Out[18]= \frac{1}{2} \frac{1}{(1 + (c - s)^2)^(3/2)} + \frac{1}{(1 + (c + s)^2)^(3/2)}
\end{verbatim}

\(^{6}\)See footnote 5.
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

\begin{verbatim}
In[19]:= c100 = Bpair /. c -> 1.0;
In[20]:= c050 = Bpair /. c -> 0.5;
In[21]:= c025 = Bpair /. c -> 0.25;
In[22]:= Plot[ { c100, c050, c025 }, {s, 0.0, 3.0},
          PlotRange -> {0.0, 2.0}, PlotStyle -> Thickness[0.005],
          AxesLabel -> {"s", "B/Bc"} ]
\end{verbatim}

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{verbatim}
In[23]:= tay = Simplify[ Series[ Bpair, { s, 0, 4 } ] ]
\end{verbatim}

\begin{verbatim}
Out[23]= 1 + \frac{3(-1 + 4c^2)s^2}{2(1 + c^2)^2} + \frac{15(1 - 12c^2 + 8c^4)s^4}{8(1 + c^2)^4} + O[s]^5
\end{verbatim}

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\footnote{Remember the structure of the storage of Taylor series as described in Section 8.8.7.}
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, of course, 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

\[
\text{In[26]} := \text{tay} = \text{Normal[tay]};
\]

\[
\text{Out[26]} = 1 + \frac{3(-1 + 4c^2)s^2}{2(1 + c^2)^2} + \frac{15(1 - 12c^2 + 8c^4)s^4}{8(1 + c^2)^4}
\]

\[
\text{In[27]} := \text{tay} /. c \rightarrow \frac{1}{2}
\]

\[
\text{Out[27]} = 1 - \frac{144}{125}s^4
\]

\[
\text{In[28]} := \text{tay1} = \% // \text{N}
\]

\[
\text{Out[28]} = 1.0 - 1.152s^4
\]

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

\[
\text{In[29]} := \text{Solve[tay1 == 0.99, s]}
\]

\[
\text{Out[29]} = \{\ldots, \{s \rightarrow 0.305237\}, \ldots\}
\]

\[
\text{In[30]} := \text{Solve[tay1==0.95, s]}
\]

\[
\text{Out[30]} = \{\ldots, \{s \rightarrow 0.456435\}, \ldots\}
\]

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 \( B_{\text{pair}} \) but restrict it to the Helmholtz case by setting \( c = 1/2 \) with the statement

\[
\text{In[31]} := B_{\text{helm}} = B_{\text{pair}} /. c \rightarrow 1/2:
\]

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 \( B_{\text{helm}} - 0.99 = 0 \) and \( B_{\text{helm}} - 0.95 = 0 \). The dependence of these expression on \( s \), however, is complicated, and the command \text{Solve} either takes a very long time or fails altogether. Thus, we draw (prematurely—see Section 14.13) on \text{Mathematica}'s command \text{FindRoot}, which uses an iterative numerical procedure to refine a supplied initial guess for the root. The above results provide us with sensible initial guesses, so we invoke the statements\(^8\)

\(^8\)In FindRoot, the first argument is the equation whose root is sought and the second is a list whose first element is...
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

In[34] := N[ Bhelm /. s -> 1/2 ]
Out[34] = 0.945824
In[35] := Quit[]

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 \( s^4 = (z/a)^4 \). 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.

13.4.6 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. Mathematica 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 \( \text{very small} \). We simply determine a Taylor expansion of the integrand with the statements

In[1] := integ = 1/Sqrt[ 1 - k^2*Sin[\[Phi]]^2 ]
Out[1] = \( \frac{1}{\sqrt{1 - k^2 \sin^2 \phi}} \)
In[2] := intapprox = Series[ integ, {k, 0, 5} ]
Out[2] = \( 1 + \frac{1}{2} \sin^2 k^2 + \frac{3}{8} \sin^4 k^4 + O[k]^6 \)

Then we insert the pre-multiplying factor \( 2/\pi \) and integrate the series on \( \phi \) with the statement

In[3] := Expand[ 2*Integrate[ intapprox, { \[Phi], 0, Pi/2 } ] / Pi ]
Out[3] = \( 1 + \frac{1}{4} k^2 + \frac{9}{64} k^4 \)

Finally, we substitute \( \sin(\theta_0/2) \) for \( k \) with the statement

the variable to be adjusted in finding the root and whose second element is an initial guess for the value of the root. With this syntax, \FindRoot\ invokes Newton’s method for finding the root. By default, \FindRoot\ strives to find the root so that the function at the root is zero to within \( \pm 10^{-6} \)—though this goal is sometimes machine and installation dependent.
13.4. EVALUATING INTEGRALS SYMBOLICALLY WITH MATHEMATICA

In[4]:= % /. k -> Sin[\[Theta]0/2]

and expand this result as a power series in \[Theta]0

In[5]:= Normal[ Series[ %, { \[Theta]0, 0, 5 } ] ]

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°, 10°, 15°, and 20°. We use the Mathematica statements

In[6]:= deg = { 5, 10, 15, 20 }; Set angles in degrees.
In[7]:= ampl = deg * (Pi/180); Convert angles to radians.
Out[8]= {1.00048, 1.00191, 1.0043, 1.00767}

Evidently, at amplitudes of 5°, 10°, 15°, and 20°, the period of the pendulum is approximately 0.05%, .2%, .4%, and .8% larger than the standard small-amplitude approximation given by the expression $2\pi\sqrt{l/g}$.

13.4.7 Fourier Coefficients for Half-Rectified Signal

Suppose we sought to express the half-rectified sine wave

$$ f(x) = \begin{cases} 0 & -l < x < 0 \\ \sin \frac{\pi x}{l} & 0 < x < l \end{cases} \quad (13.54) $$

in a Fourier series. In accordance with Eq. (13.51), the coefficients would then be given by the integrals

$$ a_0 = \frac{1}{l} \int_{-l}^{0} \sin \frac{\pi x}{l} \, dx \quad; \quad a_n = \frac{1}{l} \int_{0}^{l} \sin \frac{\pi x}{l} \cos \frac{n\pi x}{l} \, dx \quad; \quad b_n = \frac{1}{l} \int_{0}^{l} \sin \frac{\pi x}{l} \sin \frac{n\pi x}{l} \, dx \quad (13.55) $$

where $n = 1, 2, 3, \ldots$. We find these coefficients quickly. For $a_0$, we submit the statement

In[1]:= a0 = Integrate[ Sin[Pi*x/l], { x, 0, l } ]/l
Out[1]= \[\frac{2}{\pi}\]

Then for $a_n$ when $n$ is an integer not equal to zero, we use the statements

In[2]:= an = Integrate[ Sin[Pi*x/l]*Cos[n*Pi*x/l], { x, 0, l } ]/l
In[3]:= an = Simplify[ an, Element[ n, Integers ] ]
Out[3]= -\[\frac{1 + (-1)^n}{\pi - n^2 \pi}\]

Note, however, that this expression for $n = 1$ is indeterminate, so we evaluate that case in isolation with the statement

$^9$Note the distinction between 1 (el) and 1 (one). Here and in the next several statements, the character is el.
In[4]:= a1 = Integrate[ Sin[Pi*x/l]*Cos[Pi*x/l], { x, 0, l } ]/l
Out[4]= 0

Finally, we find $b_n$ with the statement

In[5]:= bn = Integrate[ Sin[Pi*x/l]*Sin[n*Pi*x/l], { x, 0, l } ]/l;
In[6]:= bn = Simplify[ %, Element[ n, Integers ] ]
Out[6]= 0

A close look at the integral giving $b_n$, however, reveals that this result cannot be correct for $b_1$, since the integrand in that case $[\sin^2(\pi x/l)]$ is decidedly positive and an integrand that is everywhere positive cannot yield zero for the integral.\(^{10}\) The formal evaluation of the integral (see the output of line In[5] on your screen) is, in fact, indeterminate at $n = \pm 1$. Rather than trying to evaluate that indeterminacy as a limit, we simply redo the integral with the statement

In[7]:= b1 = Integrate[ Sin[Pi*x/l]^2, { x, 0, l } ]/l
Out[7]:= 1/2

In total, we find that

\[
a_0 = \frac{2}{\pi} ; \quad a_1 = 0 ; \quad a_n = -\frac{1 + (-1)^n}{\pi(n^2 - 1)} n > 1 ; \quad b_1 = \frac{1}{2} ; \quad b_n = 0, n > 1 \quad (13.56)
\]

With these values for the Fourier coefficients, we assemble the Fourier series in accordance with Eq. (13.50) to find that

\[
f(x) = \frac{1}{\pi} - \sum_{n=2}^{\infty} \frac{1 + (-1)^n}{\pi(n^2 - 1)} \cos \frac{n\pi x}{l} + \frac{1}{2} \sin \frac{\pi x}{l}
\]

\[
= \frac{1}{\pi} + \frac{1}{2} \sin \frac{\pi x}{l} - \frac{2}{\pi} \sum_{m=1}^{\infty} \frac{1}{4m^2 - 1} \cos \frac{2m\pi x}{l}
\]

\[
= \frac{1}{\pi} + \frac{1}{2} \sin \frac{\pi x}{l} - \frac{2}{\pi} \left( \frac{1}{3} \cos \frac{2\pi x}{l} + \frac{1}{15} \cos \frac{4\pi x}{l} + \frac{1}{35} \cos \frac{6\pi x}{l} + \ldots \right) \quad (13.57)
\]

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

\[
f_n(x) = \frac{1}{\pi} + \frac{1}{2} \sin \frac{\pi x}{l} - \frac{2}{\pi} \sum_{m=1}^{n} \frac{1}{4m^2 - 1} \cos \frac{2m\pi x}{l} \quad (13.58)
\]

which approaches $f(x)$ as $n \to \infty$. Introducing the variable $\bar{x} = x/l$ (and then dropping the overbar), we then generate graphs for various values of $n$. First, we define a function from which the coefficients $a_m$ can be evaluated for various $m$ with the statement

In[8]:= a[m_] := an /. n -> 2*m
\(^{10}\)Note also that Mathematica gave us no warning that the result displayed was potentially incorrect for $n = \pm 1$. 
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. This graph was created by Mathematica.

Then, we submit the statement

\[
\text{In}[9]:= f[x_,n_]:= \frac{1}{\pi} + \frac{\sin(\pi x)}{2} + \sum_{m=1}^{n} a[m] \cos(2m\pi x)
\]

to define a function for evaluating the truncated series as a function of \( x \) for selected \( n \). Finally, we plot graphs of the truncated series for \( n = 1, 2, 4, \) and 8 with the statements

\[
\text{In}[10]:= \text{Plot}[f[x,1],\{x,-3.0,3.0\},\text{PlotRange}\rightarrow\{-0.5,1.5\},
\text{PlotStyle}\rightarrow\text{Thickness}[0.005],\text{AxesLabel}\rightarrow\{"x","f(x,1)"\}] \\
\text{In}[11]:= \text{Plot}[f[x,2],\{x,-3.0,3.0\},\text{PlotRange}\rightarrow\{-0.5,1.5\},
\text{PlotStyle}\rightarrow\text{Thickness}[0.005],\text{AxesLabel}\rightarrow\{"x","f(x,2)"\}] \\
\text{In}[12]:= \text{Plot}[f[x,4],\{x,-3.0,3.0\},\text{PlotRange}\rightarrow\{-0.5,1.5\},
\text{PlotStyle}\rightarrow\text{Thickness}[0.005],\text{AxesLabel}\rightarrow\{"x","f(x,4)"\}] \\
\text{In}[13]:= \text{Plot}[f[x,8],\{x,-3.0,3.0\},\text{PlotRange}\rightarrow\{-0.5,1.5\},
\text{PlotStyle}\rightarrow\text{Thickness}[0.005],\text{AxesLabel}\rightarrow\{"x","f(x,8)"\}] \\
\text{In}[14]:= \text{Quit}[]
\]

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.

13.5.1 Newton-Cotes Quadrature

One family of quadrature formulae can be deduced by starting with the recognition that the definite integral
\[ A = \int_a^b f(x) \, dx \]  
(13.59)
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 \( x_0 = a \), \( x_1 = a + \Delta x \), \( x_2 = a + 2 \Delta x \), \ldots, \( x_i = a + i \Delta x \), \ldots, \( x_N = b \), and let \( f(x_0) = f(a) = f_0 \), \( f(x_1) = f_1 \), \ldots, \( f(x_i) = f_i \), \ldots, \( f(x_N) = f(b) = f_N \). To deduce the simplest quadrature formula, we approximate the area of each resulting strip by the area of a rectangle whose height is the value of \( f(x) \) at the left end of the strip [Fig. 13.8(a)]. Thus
\[
\int_a^b f(x) \, dx \approx f_0 \Delta x + f_1 \Delta x + \cdots + f_i \Delta x + \cdots + f_{N-1} \Delta x
\]
(13.60)
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
\[
\int_a^b f(x) \, dx \approx M_N = \left( f_{1/2} + f_{3/2} + f_{5/2} + \cdots + f_{N-3/2} + f_{N-1/2} \right) \Delta x
\]
(13.61)
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,
\[
\int_a^b f(x) \, dx \approx T_N = \frac{1}{2} (f_0 + f_1) \Delta x + \frac{1}{2} (f_1 + f_2) \Delta x + \cdots + \frac{1}{2} (f_{N-1} + f_N) \Delta x
\]
\[
= \left( \frac{1}{2} f_0 + f_1 + f_2 + \cdots + f_i + \cdots + f_{N-1} + \frac{1}{2} f_N \right) \Delta x
\]
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).

\[ \int_a^b f(x) \, dx \approx S_N = \frac{1}{3} \left( f_0 + 4f_1 + 2f_2 + 4f_3 + \cdots + 2f_{N-2} + 4f_{N-1} + f_N \right) \Delta x \]  

which, as with the midpoint rule, is 100% accurate when \( f(x) \) is linear, i.e., when \( f(x) \) is a polynomial of the same order as the one used to approximate the function.

A final (for here) and still better approximation is obtained if we pair the strips—which then requires \( N \) to be even—and approximate the area of each pair by the area under the parabola fitted to the values of \( f(x) \) at the three points defining the pair. The result,

\[ \int_a^b f(x) \, dx \approx S_N = \frac{1}{3} \left( f_0 + 4f_1 + 2f_2 + 4f_3 + \cdots + 2f_{N-2} + 4f_{N-1} + f_N \right) \Delta x \tag{13.63} \]

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

\[ T_1 = T_{2^0} = \frac{f(a) + f(b)}{2} (b - a) \tag{13.64} \]

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

\[ T_2 = T_{2^1} = \frac{f(a)}{2} + f(x_1) + \frac{f(b)}{2} \left( \frac{b - a}{2} \right) = \frac{1}{2} T_1 + f(x_1) \frac{(b - a)}{2} \]

\[ = \frac{1}{2} T_1 + f \left( a + \frac{b - a}{2} \right) \frac{(b - a)}{2} \tag{13.65} \]
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

$$T_4 = T_{2^2} = \left[ \frac{f(a)}{2} + f(x_2) + f(x_1) + f(x_3) + \frac{f(b)}{2} \right] \frac{(b - a)}{4}$$

$$= \frac{1}{2} T_2 + \left[ f(x_2) + f(x_3) \right] \frac{(b - a)}{4}$$

$$= \frac{1}{2} T_2 + \left[ f \left( a + \frac{b-a}{4} \right) + f \left( a + 3 \frac{b-a}{4} \right) \right] \frac{(b - a)}{4}$$

(13.66)

Yet again, introducing

$$x_4 = a + \frac{1}{8} (b - a) \quad x_5 = a + \frac{3}{8} (b - a)$$

$$x_6 = a + \frac{5}{8} (b - a) \quad x_7 = a + \frac{7}{8} (b - a)$$

(13.67)

and continuing one more step, we find that

$$T_8 = T_{2^3} = \frac{1}{2} T_4 + \left[ f(x_4) + f(x_5) + f(x_6) + f(x_7) \right] \frac{(b - a)}{8}$$

$$= \frac{1}{2} T_4 + \left[ f \left( a + \frac{b-a}{8} \right) + f \left( a + \frac{3b-a}{8} \right) + f \left( a + \frac{5b-a}{8} \right) \right.$$

$$f \left( a + \frac{7b-a}{8} \right) \left] \frac{(b - a)}{8} \right.$$

(13.68)

and, in general, with

$$x_{2N+j} = a + \frac{(2j+1)(b-a)}{2^{N+1}} \quad 0 \leq j < 2^{N+1}$$

(13.69)

that

$$T_{2^{N+1}} = \frac{1}{2} T_{2^N} + \left[ f \left( a + \frac{b-a}{2^{N+1}} \right) + f \left( a + \frac{3b-a}{2^{N+1}} \right) + \ldots \right.$$

$$+ f \left( a + \frac{(2^N - 1)b-a}{2^{N+1}} \right) \left] \frac{(b - a)}{2^{N+1}} \right.$$

(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
\[ T_8 = \left[ \frac{1}{2} f(a) + f(x_4) + f(x_2) + f(x_5) \right. \]
\[ + f(x_1) + f(x_6) + f(x_3) + f(x_7) + \frac{1}{2} f(b) \]
\[ \left. \right] \frac{b - a}{8} \]  
(13.72)

Note, in particular, the combination
\[ T_8 - T_4 = \left[ f(a) + 4f(x_4) + 2f(x_2) + 4f(x_5) + 2f(x_1) \right. \]
\[ + 4f(x_6) + 2f(x_3) + 4f(x_7) + f(b) \left( \frac{b - a}{8 \times 3} \right) \]  
(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
\[ \frac{4T_8 - T_4}{3} = \left[ f(a) + 4f(x_4) + 2f(x_2) + 4f(x_5) + 2f(x_1) \right. \]
\[ + 4f(x_6) + 2f(x_3) + 4f(x_7) + f(b) \left( \frac{\Delta x}{3} \right) \]  
(13.74)

which—\textit{mirabile dictu}—we recognize as Simpson’s rule for evaluating the integral with 8 divisions of the interval! We conclude that
\[ S_8 = \frac{4T_8 - T_4}{3} \]  
(13.75)

More generally, we could also conclude that
\[ S_{2n} = \frac{4T_{2n} - T_n}{3} \]  
(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 \textit{four} values \(T_n, T_{2n}, T_{4n},\) and \(T_{8n}\). We could then use Eq. (13.76) to generate the \textit{three} values \(S_{2n}, S_{4n},\) and \(S_{8n}\). In Romberg integration, we next generate a pair of still more accurate values from the formulae \(X_{4n} = (16S_{4n} - S_{2n})/15\) and \(X_{8n} = (16S_{8n} - S_{4n})/15\); and then we generate a further improved value from the formula \(Y_{8n} = (64X_{8n} - X_{4n})/63\). This process can, of course, be continued indefinitely—though we rarely have to go even as far as we have described.

### 13.5.3 Assessing Error

Numerical evaluations, of course, only approximate the integral. Two distinctly different sorts of errors can occur. \textit{Truncation errors} arise because the integral has been approximated by a finite sum; \textit{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, 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.

---

11 We shall make this criterion a bit more explicit in the next paragraphs.
12 We shall see in later sections how we might decide when roundoff has started to be significant.
13 These formulae may be found in several sources, including Philip J. Davis and Philip Rabinowitz, Numerical Integration (Blaisdell Publishing Company, Waltham, MA, 1967), pp. 17, 20; and Milton Abramowitz and Irene A. Stegun, Handbook of Mathematical Functions (National Bureau of Standards Applied Mathematics Series 55, June, 1964), pp. 885ff.
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 $S_2$, $S_4$, $S_8$, ..., 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

\[
\int_a^b g(t) \, dt = \sum_{i=0}^m \int_{t_i}^{t_{i+1}} g(t) \, dt
\]  

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

\[
\int_a^b g(t) \, dt = \sum_{i=1}^m \int_{t_i}^{t_{i+1}} g(t) \, dt = \sum_{i=1}^m \frac{\Delta t_i}{2} \int_{-1}^1 g(t_i^{\text{mid}} + \frac{\Delta t_i}{2} x) \, dx = \sum_{i=1}^m \frac{\Delta t_i}{2} \int_{-1}^1 f_i(x) \, dx
\]  

where \( f_i(x) = g(t_i^{\text{mid}} + \frac{\Delta t_i}{2} x) \). 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
\[ w_1 x_1 = -w_2 x_2 \quad \text{and} \quad w_1 x_1^3 = -w_2 x_2^3 \]
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
\[ \frac{2}{3} = (w_1 + w_2)x_2^2 = 2x_2^2 \implies x_2^2 = \frac{1}{3} \] from which we also conclude that \[ x_1^2 = \frac{1}{3}. \] Thus,
\[ x_1 = -\frac{1}{\sqrt{3}} \quad \text{and} \quad x_2 = \frac{1}{\sqrt{3}} \]
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 \[ \int_{-1}^{1} f(x) \, dx = f \left( -\frac{1}{\sqrt{3}} \right) + f \left( \frac{1}{\sqrt{3}} \right) \] 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.\(^{14}\) Returning to the original function and variables as laid out in Eq. (13.85), we finally find that Eq. (13.96) supports the expression
\[ \int_{a}^{b} g(t) \, dt = \sum_{i=1}^{m} \frac{\Delta t_i}{2} \left[ g \left( t_{i}^{\text{mid}} - \frac{\Delta t_i}{2} \frac{1}{\sqrt{3}} \right) + g \left( t_{i}^{\text{mid}} + \frac{\Delta t_i}{2} \frac{1}{\sqrt{3}} \right) \right] \]
\[ = \sum_{i=1}^{m} \frac{\Delta t_i}{2} \sum_{j=1}^{2} w_j g \left( t_{i}^{\text{mid}} + \frac{\Delta t_i}{2} x_j \right) \]
(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
\[ \int_{-1}^{1} f(x) \, dx = 0.23692689 f(-0.90617985) + 0.47862867 f(-0.53846931) \]
\[ + 0.56888889 f(0.00000000) \]
\[ + 0.47862867 f(0.53846931) + 0.23692689 f(0.90617985) \] which is 100% accurate for polynomials of the ninth-degree or lower, is among the most popular of the formulae in this class. For the sake of later examples, we note that, in this expression
\[ w_1 = 0.23692689 \quad \text{and} \quad x_1 = -0.90617985 \]
\[ w_2 = 0.47862867 \quad x_2 = -0.53846931 \]
\[ w_3 = 0.56888889 \quad x_3 = +0.00000000 \]
\[ w_4 = 0.47862867 \quad x_4 = +0.53846931 \]
\[ w_5 = 0.23692689 \quad x_5 = +0.90617985 \]  \(^{14}\) 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
\[
\int_a^b g(t) \, dt = \sum_{i=1}^m \frac{\Delta t_i}{2} \sum_{j=1}^5 w_j g\left(t_i^{\text{mid}} + \frac{\Delta t_i}{2} x_j\right) \tag{13.100}
\]

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

\[
L_N(x_i) = 0 \quad \text{and} \quad w_i = \frac{2}{(1 - x_i^2)(dL_N(x_i)/dx)^2} \tag{13.101}
\]

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 \textit{Gauss-Legendre formulae}.

### 13.9 Evaluating Integrals Numerically with PYTHON

\textit{Note:} All PYTHON program (.py) files referred to in this chapter are available in the directory \$\text{HEAD/python}, where (as defined in the \textit{Local Guide}) \$\text{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 PYTHON's default search path. If so, the files can be found by PYTHON 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 PYTHON's elementary commands as described in Section 13.9.1 or, more simply, by invoking one of the built-in routines as described in Section 13.9.2.

#### 13.9.1 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

\[
I = \text{erf}(1) = \frac{2}{\sqrt{\pi}} \int_0^1 e^{-x^2} \, dx \tag{13.103}
\]


\[16\text{A more general integral that can be approached with the techniques of this subsection has the form } \int_a^b f(x) \, w(x) \, dx, \text{ where } w(x) \text{ is a weighting function. In the case we dealt with, } w(x) = 1 \text{ and, in our rescaling, the interval became the interval from } -1 \text{ to } 1. \text{ That the Legendre polynomials } L_i(x) \text{ are orthogonal on the interval } -1 < x < 1 \text{ with weight } w(x) = 1 \text{ is part of the reason that the roots of these polynomials ultimately emerged as important. For other weight functions and other intervals, a different set of polynomials would have played the role of the Legendre polynomials. Thus, there are several different types of Gaussian quadrature, each specific to a particular weight and basic interval.}\]
13.9. EVALUATING INTEGRALS NUMERICALLY WITH PYTHON

Table 13.1: Values of erf(1.0) obtained by the trapezoidal rule and a user-constructed program in PYTHON. Values were determined by double-precision calculations and all resulting digits are shown, even though not all are significant.

<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>0.77174332258</td>
<td>64</td>
<td>0.842683902045</td>
<td>4096</td>
<td>0.842700788826</td>
</tr>
<tr>
<td>2</td>
<td>0.825262955597</td>
<td>128</td>
<td>0.842696570249</td>
<td>8192</td>
<td>0.842700791919</td>
</tr>
<tr>
<td>4</td>
<td>0.838367777441</td>
<td>256</td>
<td>0.842699737276</td>
<td>16384</td>
<td>0.842700792692</td>
</tr>
<tr>
<td>8</td>
<td>0.841619221245</td>
<td>512</td>
<td>0.842700529031</td>
<td>32768</td>
<td>0.842700792885</td>
</tr>
<tr>
<td>16</td>
<td>0.842430505490</td>
<td>1024</td>
<td>0.842700792690</td>
<td>65536</td>
<td>0.842700792934</td>
</tr>
<tr>
<td>32</td>
<td>0.842633227681</td>
<td>2048</td>
<td>0.842700776455</td>
<td>131072</td>
<td>0.842700792946</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\(^\text{17}\):

```python
import numpy as np
n = 4
x = np.linspace( 0.0, 1.0, n+1 )
f = 2.0*np.exp(-x**2)/np.sqrt(np.pi)
y = 0.5*f[0]
for i in np.arange(1,n):
y = y + f[i]
y = y + 0.5*f[n]
int = (1.0/n) * y
print( int )
0.838367777441
```

Import needed modules.
Set number of segments.
Set values of \( x \).
Evaluate integrand.
Initialize accumulator to first term.
Add middle terms.
Add last term.
Rescale; display result.

where, of course, not all of the digits are significant. 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 13.1. On the basis of this succession of numbers, however, we conclude that, to six decimal places, the value of the integral is 0.842701. The trapezoidal rule took 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 \( n = 512 \)) the result stabilizes—at least to the first six or seven digits after the decimal point—and further increase of \( n \) makes almost no difference in the value to that precision. 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 \( n = 131072 \). Were we 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, 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

\[
S_2 = \frac{4 \times T_2 - T_1}{3} = \frac{4 \times 0.825262955597 - 0.77174332258}{3} = 0.8431028300433333
\]

\(^\text{17}\)Alternatively, as long as \( n \geq 2 \), the fifth through eighth lines in this code can be replaced with the single statement \( y = -0.5*f[0] + \text{sum}( f ) - 0.5*f[n] \).
and

\[
S_4 = \frac{4\cdot T_4 - T_2}{3} = \frac{4\cdot 0.838367777441 - 0.825262955597}{3} = 0.842736051389
\]  \hspace{1cm} (13.105)

Continuing this process of Richardson extrapolation, we find the values

\[
S_8 = 0.842700335846, \quad S_{16} = 0.842700801745, \quad \text{and} \quad S_{32} = 0.842700801745.
\]

Because \(S_{16}\) and \(S_{32}\) 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 17 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 need to 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\(^{18}\)

\[
\begin{align*}
n & = 2; \quad \text{Set number of segments.} \\
x & = \text{np.linspace}(0.0, 1.0, n+1); \quad \text{Set values of } x. \\
f & = 2.0*\text{np.exp(-x**2)}/\text{np.sqrt(np.pi)}; \quad \text{Evaluate integrand.} \\
w & = \text{np.zeros}(n+1); \quad \text{Set first and last weights to 1.} \\
w[0] & = 1.0; \quad w[n] = 1.0; \quad \text{Set proper weights to 4.} \\
\text{for i in np.arange(1,n,2): } w[i]=4.0; \quad \text{Set remaining weights to 2.} \\
y & = 0.0; \quad \text{Initialize accumulator.} \\
\text{for i in np.arange(0,n): } y = y + w[i]*f[i]; \quad \text{Add each strip with proper weight.} \\
\text{int} & = (1.0/n)*y/3.0; \quad \text{Rescale; display result.} \\
\text{print( int )} & = 0.843102830043
\end{align*}
\]

where, again, not all digits are significant. If this process is repeated with \(n = 4\), the end result is 0.842736051389. With \(n = 32\) and \(n = 64\), this process yields the values 0.842700801745 and 0.8427007935, 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 \(g(t) = 2e^{-x^2}/\sqrt{\pi}\) and we divide the interval \(0 \leq x \leq 1\) into \(m = 4\) segments of equal width, we might invoke the PYTHON statements

\[
\begin{align*}
\text{pts} & = [-1.0/\text{np.sqrt(3.0)}, 1.0/\text{np.sqrt(3.0)}]; \quad \text{Set evaluation points } x_1 \text{ and } x_2. \\
w & = [1.0, 1.0]; \quad \text{Set weights } w_1 \text{ and } w_2. \\
a & = 0.0; \quad b = 1.0; \quad \text{Set limits.}
\end{align*}
\]

\(\) to set the evaluation points, weights, and limits. Then we would execute the statements

\[
\begin{align*}
m & = 4; \quad \text{Set number of segments.} \\
x & = \text{np.linspace(a, b, m+1)}; \quad \text{Set values dividing segments.} \\
xmid & = \text{np.zeros(m)}; \quad \text{Prepare list for midpoint values.} \\
\text{for i in np.arange(0,m): } xmid[i] & = (x[i]+x[i+1])/2.0; \quad \text{Evaluate midpoints.}
\end{align*}
\]

\(\) 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

\[\]

\(^{18}\)Alternatively, 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 \(y = f \cdot \text{transpose}(w)\).
13.9. EVALUATING INTEGRALS NUMERICALLY WITH PYTHON

\[ \int = \frac{(b-a)}{m} \]
\[ \text{int} = 0.0 \]
for \( i \) in np.arange(0,m):
  \[ \text{fcts} = 0.0 \]
  for \( j \) in [0,1]:
    \[ \text{fcts} = \text{fcts} + w[j]*\exp((-xmid[i]+dx*pts[j]/2.0)**2) \]
  \[ \text{int} = \text{int} + \text{fcts} \]
\[ \text{int} = \text{dx*int/np.sqrt(np.pi)} \]
print( \text{int} )
0.842699298102

finding that the two-point Gaussian formula with two 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 0.842441892523, 0.842677323863, 0.842699298102, and 0.842700699207, 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 in Eq. (13.99). We begin by invoking the statements

\[ \text{pts} = \{-0.90617985, -0.53846931, 0.0, 0.53846931, 0.90617985\} \]
\[ \text{w} = \{0.23692689, 0.47862867, 0.56888889, 0.47862867, 0.23692689\} \]
\[ a = 0.0; b = 1.0 \]

to set the evaluation points \( x_1, \ldots, x_5 \), the weights \( w_1, \ldots, w_5 \) and the limits. Then, we prepare to evaluate the integral with the statements

\[ \text{m} = 4 \]
\[ \text{dx} = \frac{(b-a)}{m} \]
\[ \text{x} = \text{np.linspace}( a, b, \text{m}+1 ) \]
\[ \text{xmid} = \text{np.zeros(m)} \]
for \( i \) in np.arange(0,m):
  \[ \text{xmid}[i]=(\text{x}[i]+\text{x}[i+1])/2.0 \]

Finally, we evaluate the sum, multiply with an overall factor whose inclusion was postponed, and display the result with the statements

\[ \text{int} = 0.0 \]
for \( i \) in np.arange(0,m):
  \[ \text{fcts} = 0.0 \]
  for \( j \) in [0,1,2,3,4]:
    \[ \text{fcts} = \text{fcts} + w[j]*\exp((-xmid[i]+dx*pts[j]/2.0)**2) \]
  \[ \text{int} = \text{int} + \text{fcts} \]
0.842700797138

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 result 0.842700789923—which agrees with the above result to seven digits—with \( m = 1 \), i.e., when the entire interval of integration is treated as a single segment!

13.9.2 Built-In Integration Routines

The \texttt{scipy.integrate} module for PYTHON contains several routines for evaluating integrals numerically. One-dimensional integrals can be evaluated numerically either by using PYTHON’s elementary commands or, more simply, by invoking one of the built-in routines. Some, specifically
evaluate one-dimensional integrals of a user-supplied function \( f(x) \) and are invoked with a statement like

\[
\text{VarName} = \text{RoutineName}( \text{Function}, \text{LowLim}, \text{UpLim} )
\]

where \( \text{VarName} \) is the variable in which the value returned will be stored, \( \text{RoutineName} \) is the name of the integration routine to be invoked,\(^{19}\) \( \text{Function} \) is a properly defined function returning the integrand, and \( \text{LowLim} \) and \( \text{UpLim} \) are the lower and upper limits of integration. As discussed in Section 13.9.10, these routines also admit a number of keywords to control tolerances and several other specific behaviors of the integration. To keep things simple initially, we will accept all defaults, which in particular stipulate absolute and relative errors of \( 1.49 \times 10^{-8} \).

Sometimes, for example with data acquired experimentally, one seeks to integrate a function defined by values in a table that provides values of the function at equally spaced values of the independent variable. The module \texttt{scipy.integrate} provides also the routines

- \texttt{scipy.integrate.trapz} and \texttt{scipy.integrate.cumtrapz}, which use the trapezoidal rule,
- \texttt{scipy.integrate.simps}, which uses Simpson’s Rule that works best with an even number of intervals (and an odd number of values) but knows how to handle the other case, and
- \texttt{scipy.integrate.romb}, which uses Romberg integration

to evaluate one-dimensional integrals when the function is supplied by a table of values. The routines \texttt{trapz}, \texttt{cumtrapz}, and \texttt{simps} are invoked with a statement like

\[
\text{VarName} = \text{RoutineName}( y, x )
\]

while the routine \texttt{romb} is invoked with a statement like

\[
\text{VarName} = \text{romb}( y, \text{dx=Space} )
\]

Here \( \text{VarName} \) is the variable in which the value returned will be stored, \( \text{RoutineName} \) is the name of the integration routine to be invoked,\(^{20}\) \( y \) is a list or an array providing values of the dependent variable at the (equally-spaced) values of the independent variable provided in the list or array \( x \), and \( \text{dx} \) is a keyword that conveys the difference between consecutive values of the independent variable.

Information about the available routines for evaluating two-, three- and n-dimensional integrals is left to the PYTHON manuals.

The optimal method to use depends on the character of the integrand. For smooth integrands, all methods will work, though some will require more calculation than others to achieve a particular accuracy. Most can deal with integrals that have infinite limits and a few will work well with integrands that have singularities. In the following sections, we illustrate the use of only a few of these options. Full details on each of these commands can be found in the PYTHON manuals and on the web.\(^{21}\)

---

\(^{19}\) The routine must, of course, have been appropriately imported before invoking it.

\(^{20}\) The routine must, of course, have been appropriately imported before invoking it.

\(^{21}\) See docs.scipy.org/doc/scipy/reference/tutorial/integrate.html and docs.scipy.org/doc/scipy/reference/integrate.html for details on all of these utilities.
The simplest of these routines—`trapz`—is straightforward and non-adaptive. It simply uses the trapezoidal rule to evaluate \( \int y \, dx \) when supplied with two one-dimensional lists or arrays \( x \) and \( y \), the first containing equally spaced values of the independent variable and the second containing values of the dependent variable at the points in \( x \). Thus, for example, we might evaluate the integral in the previous section via the trapezoidal rule with 16 segments with the statements

```python
import numpy as np
import scipy.integrate as sp
x = np.linspace( 0.0, 1.0, 17 )
y = 2.0*np.exp(-x**2)/np.sqrt(np.pi)
I = sp.trapz( y, x ); print( I )
0.8424305054902326
I = sp.simps( y, x ); print( I )
0.842700933572054
I = sp.romb( y, dx=1.0/16.0 ); print( I )
0.8427007932686706
```

As invoked here, the routine `trapz` has, of course, yielded \( T_{16} \)—and this result agrees with previous evaluations of the integral by this method with 16 divisions. The routine `cumtrapz` differs from `trapz` simply by printing out the accumulating value of the integral as each next step is added to the result, and it may produce a substantial volume of output. The result from `trapz` is accurate only to the third digit; those from `simps` and `romb` are accurate at least through the sixth digit.

Before invoking any routine that integrates a defined function, we must create a function py-file that returns the integrand as a function of a single scalar argument (the integration variable). The construction of such files is described in Section 5.7. For the integral in Eq. (13.41), for example, the function py-file might be

```python
def gausint(t):
    # Gausint - Defines Gaussian lineshape
    # Gausint defines the integrand for evaluating the error function.
    tmp1 = 2.0/np.sqrt(np.pi)
    y = tmp1*np.exp(-t**2)
    return y
```

Having stored the py-file in the user’s default directory with a name identical to that of the function name and with the file extension .py (here `gausint.py`) and using defaults for all keywords, we would evaluate the integral and display its value with a statements

```python
execfile('gausint.py') or exec(open('gausint.py').read())
import numpy as np
import scipy.integrate as sp
int = sp.quad( gausint, 0.0, 1.0 )
print( int )
(0.8427007929497149, 9.355858232026503e-15)
int = sp.fixed_quad( gausint, 0.0, 1.0 )
print( int )
(0.8427007861273327, None)
```

---

22 Inclusion of parameters in the definition of the function will be discussed in Section 13.9.8.
23 `quad` returns a tuple containing the value of the integral and an estimate of the error in that value; `fixed_quad` returns a tuple containing the value of the integral and a fixed value `None`; `quadrature` returns a tuple containing the value of the integral and the difference between the last two estimates of the integral; and `romberg` returns only the value of the integral.
int = sp.quadrature( gausint, 0.0, 1.0 )
print( int )
    (0.8427007930374221, 6.910089389577934e-09)
int = sp.romberg( gausint, 0.0, 1.0 )
print( int )
    0.842700792949508

With the default tolerance of $1.49 \times 10^{-8}$, we have reason to believe that these results are all accurate to about the seventh digit, though quad claims an accuracy much better than that and romberg implies an accuracy a bit better than that. For example, if we believe the change reported by quadrature, we might presume the value of the integral to be $0.842700793 \pm 0.000000007$.

## 13.9.3 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 M-file

```python
def moment( lamb ):
    # MOMENT - Defines integrand for moment of inertia
    # MOMENT defines the integrand for evaluating the moment of
    # inertia of a semicircular plate.
    tmp = lamb**2
    y = 4.0*tmp*np.sqrt(1.0-tmp)/np.pi
    return y
```

and store it in a file named moment.py. Then, the simple statements

```python
import numpy as np
import scipy.integrate as sp
execfile('moment.py') or exec(open('moment.py').read() )
I = sp.quad( moment, 0.0, 1.0 ); print( I )
    (0.2499999999999999478, 1.003955966354808e-09)
I = sp.quadrature( moment, 0.0, 1.0 ); print( I )
    ... AccuracyWarning: maxiter (50) exceeded. Latest difference = 8.915380e-07
    (0.250001440787829, 8.915380134144613e-08)
I = sp.quadrature( moment, 0.0, 1.0, maxiter=100 ); print( I )
    (0.2500003690920863, 1.4289524896682337e-08)
```

Here, prompted by the first output from quadrature, we have (prematurely—see Section 13.9.10) invoked the keyword maxiter to increase the allowed number of iterations and succeeded in suppressing the warning first displayed. These results are clearly in agreement with one another and with those obtained by other methods in this chapter, provided we don’t believe more than five or six digits, though the value given by quad implies that $I = 0.25000000000 \pm 0.000000001$.

If we didn’t know, however, what the “exact” value should be, we would have to be more careful about interpreting the above numerical result. We could, for example, exploit the simpler, non-adaptive command trapz with the statements

```python
import numpy as np
import scipy.integrate as sp
execfile('moment.py') or exec(open('moment.py').read() )
I = sp.quad( moment, 0.0, 1.0 ); print( I )
    (0.2499999999999999478, 1.003955966354808e-09)
I = sp.quadrature( moment, 0.0, 1.0 ); print( I )
    ... AccuracyWarning: maxiter (50) exceeded. Latest difference = 8.915380e-07
    (0.250001440787829, 8.915380134144613e-08)
I = sp.quadrature( moment, 0.0, 1.0, maxiter=100 ); print( I )
    (0.2500003690920863, 1.4289524896682337e-08)
```
13.9. EVALUATING INTEGRALS NUMERICALLY WITH PYTHON

n = 10
x = np.linspace(0.0, 1.0, n+1)
y = 4.0 * x**2 * np.sqrt(1.0-x**2)/np.pi
I = trapz(x, y)
I = 0.2384963329797492

to evaluate the desired integral via the trapezoidal rule with 10 segments in the interval \(0 \leq x \leq 1.\)
We can repeat this process for larger values of \(n\), finding the values

<table>
<thead>
<tr>
<th>(n)</th>
<th>(I)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.24962670804382944</td>
</tr>
<tr>
<td>1000</td>
<td>0.2499881660155479</td>
</tr>
<tr>
<td>10000</td>
<td>0.24999962568360487</td>
</tr>
<tr>
<td>100000</td>
<td>0.2499999881627824</td>
</tr>
<tr>
<td>1000000</td>
<td>0.24999999962567337</td>
</tr>
</tbody>
</table>

Since, as discussed in Section 13.5.3, the error in the trapezoidal rule decreases in inverse proportion
to the square of \(n\), each new step in this sequence should yield an improvement of a factor of 100
(two decimal digits) in precision. By replacing the automatic criterion with a personal examination,
we can be more confident that the value of this integral is approaching 0.25000.\(^{24}\)

13.9.4 Quantum Probabilities

In Section 13.9.2, 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

\[
P(|x| > |x_{\text{turn}}|) = 1.0 - \text{erf}(1.0) = 1.0 - 0.842701 = 0.157299 \quad (13.106)
\]

We conclude that, in a bit over 15% of the measurements, the quantum oscillator will be found in
the classically forbidden region!

13.9.5 Integrals as Functions of the Upper Limit

To evaluate an integral as a function of a parameter, the routine used 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

\[
g(x) = \int_0^x f(t) \, dt \quad (13.107)
\]

We begin by constructing a py-file to define the integrand \(f(t)\), say integ.py. Then, supposing
that we want to evaluate the integral as a function of its upper limit \(x\) for \(a \leq x \leq b\), we invoke the PYTHON statements\(^{25,26}\)

\(^{24}\)Remember that PYTHON does its internal arithmetic in double precision, so we have probably not yet begun to
be affected by problems from internal roundoff.

\(^{25}\)We might, of course, use fixed_quad or quadrature instead of quad in this example. If we used romberg, the
characters \[0\] at the end of the line beginning \(g[i]\) would not be there.

\(^{26}\)We assume that numpy and scipy.integrate have been imported as np and sp, respectively.
N = (appropriate value)
x = np.linspace( a, b, N+1 )
g = np.zeros(N+1)
for i in np.arange(0, N+1):
g[i] = sp.quad( integ, 0, x[i] )[0]

Set number of segments in interval.
Set values for upper limit.
Create list for values of integral.
Evaluate integral for each upper limit in x,
using quad and accepting the default toler-
ance.

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 list \( g \) and can be further processed as desired. In particular, the values in \( g \) could be plotted versus the values in \( x \) to generate a graph of \( g \) versus \( x \).27

13.9.6 The Error Function

To evaluate the error function as given by Eq. (13.26), we begin by constructing the py-file giving the integrand. The function gausint.m defined in Section 13.9.2 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 PYTHON statements

```python
import matplotlib.pyplot as plt
N = 30
x = np.linspace( 0.0, 3.0, N+1 )
erf = np.zeros(N+1)
for i in np.arange(0,N+1):
erf[i] = sp.quad( gausint, 0.0, x[i] )
plt.plot( x, erf, linewidth=4,
        color='black' )
plt.title('Error Function', fontsize=20 )
plt.xlabel( 'x', fontsize=16 )
plt.ylabel( 'erf(x)', fontsize=16 )
plt.tick_params(labelsize=12)
plt.grid()
plt.show()
```

Import needed module.
Set number of segments.
Specify values of upper limit.
Prepare list for dependent variable.
Evaluate integral for each upper limit in \( x \),
accepting defaults for keywords.
Plot error function.
Title graph.
Label axes.
Adjust tick label size.
Turn on grid.
Display plot.

The resulting graph is shown in Fig. 13.10.

13.9.7 The Cornu Spiral

To determine the Cornu spiral, we use quad 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 py-files

```python
def spiralc(t):
    # SPIRALC - returns cosine integrand for Cornu spiral.
    # SPIRALC defines the integrand for the integral giving the
    # vertical coordinate of the Cornu spiral.
    y = np.cos( np.pi * t**2 / 2.0 )
    return y
```

27This 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.
def spirals(t):
    # SPIRALS - returns sine integrand for Cornu spiral.
    # SPIRALS defines the integrand for the integral giving the
    # horizontal coordinate of the Cornu spiral.
    y = np.sin( np.pi * t**2 / 2.0 )
    return y

to provide the two integrands, storing them in the user's default directory in the same file with the
name spiral.py. For definiteness, we elect to explore the spiral over the range $-5 \leq u \leq 5$, though
we evaluate only 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 generate the necessary values of the upper limit and
evaluate the integrals with the statements

```
execfile('spiral.py') or
exec(open('spiral.py').read() )
N = 100
u = np.linspace( 0.0, 5.0, N+1 )
CP = np.zeros(101); SP = np.zeros(101)
CN = np.zeros(101); SN = np.zeros(101)
for i in np.arange(0,N+1):
    CP[i]=sp.quad( spiralc, 0.0, u[i] )[0]
    SP[i]=sp.quad( spirals, 0.0, u[i] )[0]
    CN[N - i] = -CP[i]
    SN[N - i] = -SP[i]
```

Finally, we assemble the variables to be plotted and plot the graph with the statements

```
C = np.append(CN, CP); S = np.append(SN, SP)
plt.plot( C, S, linewidth=3, color='black' )
```

Define functions. See Section 5.7.
Set number of segments.
Specify values of upper limit.
Prepare lists for values.
Evaluate/store integrals for each limit.
Concatenate values for positive and negative limits.
Plot graph of $S$ versus $C$. 

Figure 13.10: The error function.
13.9.8 Integrals as Functions of an Internal Parameter

The situation in which an integral of interest is a function of a parameter \textit{in the integrand} is more difficult because we must somehow sneak the parameters in the integrand \textit{through} the integration routine and into the function called by those routines to evaluate the integrand. One technique for achieving this objective is to exploit global variables as discussed in Section 5.7.3. Using global variables will work with all of PYTHON’s routines for integrating specified functions.\footnote{Passing parameters is not an issue when tabulated data are to be integrated.} Conveniently, these routines offer an alternative. Calls to these routines can take advantage of the keyword \texttt{args}. The full structure of statements involving these commands is

\begin{verbatim}
ValName = RoutineName( Function, LowLim, UpLim, args=... )
\end{verbatim}

Here—beyond the quantities already defined—the keyword \texttt{args} has as its target a tuple that provides arguments to be passed to the function evaluating the integrand. That function must, of course, be defined with those parameters as additional arguments. As has been previously mentioned, if the tuple assigned to \texttt{args} has only one value, that value must be followed by a seemingly irrelevant comma, e.g., \texttt{args = (P1,)}.\footnote{This sequence of statements actually has two glitches. First, it is computationally inefficient in the way described in footnote 27. Second, and less significantly, the values of $S(0)$ and $C(0)$ appear twice in the vectors finally plotted.}
To invoke this feature, we must do two things. First, we must define the integrand with the parameters as additional arguments to the function. The appropriate file would then have the general format

```python
def FunctionName( x, p1, p2, p3, ... ):
    # Explanatory comments.
    ...
    Statements to evaluate integrand, using p1, p2, p3, ... for the parameters.
    ...
    y = integrand
    return y
```

Here `FunctionName` is the user-assigned name for the function returning the integrand, `x` is the independent variable, `p1`, `p2`, `p3`, ... are the parameters, and `integrand` is the finally calculated value of the integrand. The file will normally be stored with the name `FunctionName.py`.

Second, having defined the py-file in this new way and stored it in the default directory, we invoke PYTHON to evaluate and display the desired integral by using, for example, `quad` with the statements

```python
p1 = ??; p2 = ??; p3 = ??; ...
q = sp.quad( FunctionName, Lowlim, UpLim, args=(p1,p2,p3) )
```

where we first assign values to however many parameters there are and then invoke `quad` (or some other integration routine), including a use of the keyword `args`.

Note that the names of the parameters need not be the same at command level as they are in the function py-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.

### 13.9.9 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 py-file

```python
def rings( phi, s ):
    # RINGS - Integrand for charged rings
    # RINGS defines the integrand whose integral gives the
    # electrostatic potential in the midplane between two
    # uniformly charged circular rings.
    tmp = np.sqrt( 2.0 ) / ( 2.0 * np.pi );
    y = tmp / np.sqrt( 2.0 - 2.0*s*np.cos(phi) + s*s )
    return y
```

We store this file with the name `rings.py`, then invoke PYTHON to evaluate and display the desired integral by using `quad`, say, with the statement

```python
q = quad( rings, a, b, args=(s,) )
```
Since we seek to explore this integral as a function of $s$ over, say, $0.0 \leq s \leq 4.0$, we must generate a list of values of $s$ and then, in a loop, invoke \texttt{quad} once for each element in that vector. We might use the statements

```python
import numpy as np
import matplotlib.pyplot as plt
import scipy.integrate as sp

s = np.linspace(0.0, 4.0, 41 )
N = s.size
V = np.zeros(N)
for i in np.arange(0,N):
    V[i] = sp.quad( rings, 0.0, 2.0*np.pi, args=(s[i],) )[0]
plt.plot( s, V, linewidth=4, color='black' )
plt.title( 'Potential in Midplane', fontsize=20)
plt.xlabel('$s/a$', fontsize=16)
plt.ylabel('$V(s)/V(0)$', fontsize=16)
plt.xlim( (0.0,4.0 ) )
plt.ylim( (0.0,1.0 ) )
plt.grid()
plt.show()
```

The resulting graph is shown in Fig. 13.12.

### 13.9.10 Keywords that Modify Function Integrators

When the default values of the keywords that control the detailed action of \texttt{quad}, \texttt{fixed_quad}, \texttt{quadrature}, and \texttt{romberg} are not appropriate to the task at hand, these keywords can be given different values. The available keywords with the default values include

- \texttt{args=()}, which provides parameters to be passed to the function defining the integrand,
Table 13.2: Applicability of keywords to integration routines.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>quad</th>
<th>fixed_quad</th>
<th>quadrature</th>
<th>romberg</th>
</tr>
</thead>
<tbody>
<tr>
<td>args</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>epsabs</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>tol</td>
<td></td>
<td></td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>epsrel</td>
<td>×</td>
<td></td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>rtol</td>
<td></td>
<td></td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>limit</td>
<td>×</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>maxiter</td>
<td></td>
<td></td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>n</td>
<td></td>
<td>×</td>
<td></td>
<td></td>
</tr>
<tr>
<td>divmax</td>
<td></td>
<td></td>
<td></td>
<td>×</td>
</tr>
</tbody>
</table>

- epsabs=1.49e-08 or tol=1.49e-08, which specifies the desired absolute error in the evaluation of the integral.
- epsrel=1.49e-08 or rtol=1.49e-08, which specifies the desired absolute error in the evaluation of the integral.
- limit=50, which sets the upper bound on the number of subintervals to be used in any adaptive routines.
- maxiter=50, which sets the maximum number of iterations before a message of non-convergence will be displayed.
- n=5, which specifies the order of a Gaussian quadrature procedure.
- divmax=10, which specifies the maximum number of extrapolations allowed

Table 13.2 conveys which of these keywords are recognized by which integration routines.

13.12 Evaluating Integrals Numerically with Mathematica

Mathematica offers at least two routes for evaluating integrals numerically. For integrals that can be evaluated analytically, we can simply embed the command \texttt{Integrate} within the command \texttt{N}. In response to the simple statement

\[
\texttt{N[ Integrate[ f, \{ x, a, b \} ] ]}
\]

where \( f \) is an expression that depends on \( x \). Mathematica first evaluates \( \int_a^b f \, dx \) symbolically and then determines the numerical value from the symbolic result. This same statement will automatically invoke numerical quadrature if a symbolic result cannot be obtained (and all quantities in the integrand have numerical values). If, however, an integral does not admit a symbolic evaluation (or we wish a numeric evaluation even when a symbolic evaluation exists), then we would instead invoke the command \texttt{NIntegrate} in a statement of the form

\[
\texttt{NIntegrate[ f, \{ x, a, b \} ]}
\]

While the defaults assumed by \texttt{NIntegrate} will frequently be acceptable, this command nonetheless possesses several options that give us the ability to override defaults. Among these options are
\begin{itemize}
\item **WorkingPrecision** (default \$MachinePrecision, often 16), which specifies the number of digits to be used for internal computations.
\item **AccuracyGoal** (default usually figured as half the value of WorkingPrecision), which specifies the absolute accuracy to be sought by stipulating the number of digits to be provided in the solution. AccuracyGoal should not exceed WorkingPrecision.
\item **PrecisionGoal** (default usually figured as half the value of WorkingPrecision), which specifies the relative accuracy sought in the final result. PrecisionGoal should not exceed WorkingPrecision.
\end{itemize}

*Note:* With AccuracyGoal set to \(a\) and PrecisionGoal set to \(p\), Mathematica strives for an absolute accuracy in a value \(x\) to be less than \(10^{-a} + 10^{-p}|x|\).

\begin{itemize}
\item **MaxRecursion** (default automatic), which specifies the maximum number of recursive refinements to be made in seeking the desired precision. This option provides a means to stop the iterations even if goals on accuracy and precision have not been met.
\item **Method** (default usually an adaptive algorithm chosen by Mathematica to be optimum for the integrand), which specifies the method to be used. Among the allowed values are GaussKronrodRule, TrapezoidalRule, and MonteCarloRule.\(^{30}\)
\end{itemize}

Remember, too, that the statement `Options[NIntegrate]` will display all options and their current default values for the indicated command. Fuller detail about these features of Mathematica can be found in the Mathematica manuals and in the on-line Help Browser.

With this feature of Mathematica, we can evaluate our standard example in many ways. We might, for example, use the statement

\begin{verbatim}
In[1]:= N[ Integrate[ 2*Exp[-t^2]/Sqrt[Pi], { t, 0, 1 } ] ]
Out[1]= 0.842701
\end{verbatim}

which will evaluate the integral analytically [obtaining erf(1)], evaluate the error function of argument 1, and finally display the result with the default precision intrinsic to the command N. That precision can, of course, be adjusted with the second argument so that, for example, the statement

\begin{verbatim}
In[2]:= N[ Integrate[ 2*Exp[-t^2]/Sqrt[Pi], { t, 0, 1 } ], 20 ]
Out[2]= 0.84270079294971486934
\end{verbatim}

will yield a result with 20 digits. Alternatively, we might use the statement

\begin{verbatim}
In[3]:= NIntegrate[ 2*Exp[-t^2]/Sqrt[Pi], {t, 0, 1} ]
Out[3]= 0.842701
\end{verbatim}

which jumps directly to numerical integration, probably via the (adaptive) Gauss-Kronrod method. To illustrate use of some options, we might invoke the statement

\begin{verbatim}
In[4]:= NIntegrate[ 2*Exp[-t^2]/Sqrt[Pi], {t, 0, 1}, Method -> TrapezoidalRule ]
Out[4]= 0.842701
\end{verbatim}

Reassuringly, these results are all consistent with one another.

\(^{30}\)A full listing of allowed values can be found by searching for NIntegrateIntegrationRules in the Wolfram Documentation.
13.12.1 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

\[ P(|x| > |x_{\text{turn}}|) = 1.0 - \text{erf}(1.0) = 1.0 - 0.842701 = 0.157299 \]  \hspace{1cm} (13.108)

which is (probably) correct to all of the digits shown. We conclude that in a bit over 15% of the measurements, the quantum oscillator will be found in the classically forbidden region!

13.12.2 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}
In[1]:= gaussian[t_] := 2*Exp[-t^2]/Sqrt[Pi]
In[2]:= uplim = Range[ 0.0, 3.0, 0.1 ];
In[3]:= For[ j=1, j<32, j++, q[j]=NIntegrate[ gaussian[t], {t, 0, uplim[[j]]} ] ]
\end{verbatim}

The values of the upper limits in the list \texttt{uplim} and the values \( q_i \) created by these statements must then be converted into a list of the coordinates of the points with a statement like

\begin{verbatim}
In[4]:= pts = Table[ { uplim[[i]], q[i] }, { i, 1, 31 } ];
\end{verbatim}

before the graph of Fig. 13.13 can be created with the statement

\begin{verbatim}
In[5]:= ListPlot[ pts, PlotJoined -> True, PlotStyle -> Thickness[0.01],
                  PlotLabel -> "Error Function", GridLines -> Automatic ]
\end{verbatim}

In this final statement, two new options have been stipulated explicitly. Setting \texttt{PlotJoined} to \texttt{True} instructs \textit{Mathematica} to connect consecutive points with straight line segments; setting \texttt{GridLines} to \texttt{Automatic} requests the drawing of full-length gridlines at each major tick mark on both axes.

13.12.3 The Off-Axis Electrostatic Potential of Two Rings

Using \textit{Mathematica}, 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

\begin{verbatim}
In[1]:= rings[\Phi_, s_] := Sqrt[2]/(2*Pi*Sqrt[2-2*s*Cos[\Phi]+s^2])
\end{verbatim}

defines the integrand, and the statements

\begin{verbatim}
In[2]:= s = Range[ 0.0, 3.0, 0.1 ];
In[3]:= For[ j=1, j<32, j++, V[j]=NIntegrate[ rings[\Phi], s[[j]] ],
            { \Phi, 0, 2*Pi } ] ]
In[4]:= pts = Table[ { s[[i]], V[i] }, { i, 1, 31 } ];
\end{verbatim}

establish a vector of the desired parameters, 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

\begin{verbatim}
In[5]:= ListPlot[ pts, PlotJoined -> True, PlotStyle -> Thickness[0.01],
                  PlotLabel -> "Off-Axis Electrostatic Potential", GridLines -> Automatic ]
\end{verbatim}

would create a graph of the electrostatic potential as a function of \( s \).
CHAPTER 13. EVALUATING INTEGRALS

Figure 13.13: Error function.

In[5]:= ListPlot[ pts, PlotRange -> { 0.0, 1.0 }, PlotJoined -> True, 
PlotStyle -> Thickness[0.01], GridLines -> Automatic, 
PlotLabel -> "Midplane Potential" ]

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\textsuperscript{31,32}

\textsuperscript{31}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.

\textsuperscript{32}Remember also that we are using implicit data typing.
Figure 13.14: Potential in midplane.

The second section evaluates the integral with the coding

\[ DX = \frac{(B-A)}{\text{FLOAT}(N)} \] ! Set size of segment
\[ VALUE = 0.5 \times \text{FUNC}(A) \] ! Compute first term
\[ \text{DO } I = 1, N-1 \! \quad \text{VALUE} = \text{VALUE} + \text{FUNC}(A + I\times DX) \! \quad \text{Add middle terms} \]
\[ \text{ENDDO} \]
\[ \text{VALUE} = \text{VALUE} + 0.5 \times \text{FUNC}(B) \] ! Add last term
\[ \text{VALUE} = \text{VALUE} \times DX \] ! Compute integral

Finally, the third section displays the resulting evaluation with the coding

\[ \text{WRITE(*, '[(X,A)') 'Integral = ', VALUE} \]

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

FUNCTION FUNC(X)
    \[ \text{FUNC} = 2.0 \times \text{EXP}(-X\times 2)/\text{SQRT}(3.14159265) \]
END
Table 13.3: Values of \( \text{erf}(1.0) \) obtained with the trapezoidal rule and the FORTRAN program \texttt{trapezoidal.f}.

\[
\begin{array}{cccccc}
\hline
n & T_n & n & T_n & n & T_n \\
\hline
1 & 0.771743 & 256 & 0.842700 & 65536 & 0.842701 \\
2 & 0.825263 & 512 & 0.842700 & 131072 & 0.842699 \\
4 & 0.838368 & 1024 & 0.842701 & 262144 & 0.842694 \\
8 & 0.841619 & 2048 & 0.842700 & 524288 & 0.842726 \\
16 & 0.842430 & 4096 & 0.842701 & 1048576 & 0.842636 \\
32 & 0.842633 & 8192 & 0.842701 & 2097152 & 0.843682 \\
64 & 0.842684 & 16384 & 0.842701 & 4194304 & 0.839947 \\
128 & 0.842696 & 32768 & 0.842701 & 8388608 & 0.847357 \\
\hline
\end{array}
\]

which can be placed either before or after the program itself. A full listing of this program, named \texttt{trapezoidal.f}, is presented in Section 13.A. Simply changing the definition of the function \texttt{FUNC} will create a program that will integrate a different function.

For the sake of an explicit example, we have defined \texttt{FUNC} in \texttt{trapezoidal.f} to return the integrand for the error function as defined in Eq. (13.26). To compile and run this program to evaluate \( \text{erf}(1.0) \), we execute the statements\(^{33}\)

\[
\begin{align*}
\text{f77} & \ -o \texttt{trapezoidal.xf} \texttt{trapezoidal.f} & \text{Create executable file.} \\
./\texttt{trapezoidal.xf} & \texttt{trapezoidal.f} & \text{Execute program.} \\
\text{Lower limit} & : 0.0 & \text{Enter controlling values.} \\
\text{Upper limit} & : 1.0 \\
\text{Number of segments:} & : 1 & \text{View output.} \\
\text{Integral} & = 0.771743 \\
\end{align*}
\]

The results for various divisions of the integration interval are shown in Table 13.3. 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.

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.\(^{34}\)

\(^{33}\)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 \texttt{./} is necessary in UNIX but may not be necessary with other operating systems. Details will be found in the Local Guide.

\(^{34}\)Details on the use of named common are discussed in Section 9.5.4; further illustration will be found in Section 11.14.
13.13. EVALUATING INTEGRALS NUMERICALLY WITH FORTRAN

Table 13.4: 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
```

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 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 13.4. 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 erf(1.0) = \( (2/\sqrt{\pi}) \int_0^1 e^{-x^2} dx \). The steps in evaluating the desired integral with this program then are

---

35 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.

36 A somewhat more elaborate demonstration program named `xtrapzd.f` can be found in the directory `$NRHEAD/recipes.f/demo/src`.
Table 13.5: 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>

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 13.5. 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 \(\text{eps}\) (default \(10^{-6}\)) in the subroutine *qtrap.f*] or when more than 20 refinements of the step size [as defined by the editable parameter \(\text{jmax}\) (default 20)] have been effected without convergence.\(^{37}\) It is invoked with a statement of the form

\[
\text{CALL QTRAP( FUNC, A, B, S )}
\]

where \(\text{FUNC}\) defines the function to be integrated, \(\text{A}\) and \(\text{B}\) are the lower and upper limits, and \(\text{S}\) 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 13.6. The steps in evaluating the desired integral with this program are

\(^{37}\)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 13.6: The FORTRAN program ERRQTRAP.F.

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

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 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.\(^{38}\)

### 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.

#### 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

\(^{38}\)See footnote 34.
Table 13.7: Values of erf(1.0) obtained with the trapezoidal rule and the C program `trapezoidal.c`.

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0.771743</td>
<td>512</td>
<td>0.842700</td>
<td>131072</td>
</tr>
<tr>
<td>4</td>
<td>0.838368</td>
<td>1024</td>
<td>0.842701</td>
<td>262144</td>
</tr>
<tr>
<td>8</td>
<td>0.841619</td>
<td>2048</td>
<td>0.842701</td>
<td>524288</td>
</tr>
<tr>
<td>16</td>
<td>0.842431</td>
<td>4096</td>
<td>0.842701</td>
<td>1048576</td>
</tr>
<tr>
<td>32</td>
<td>0.842633</td>
<td>8192</td>
<td>0.842701</td>
<td>2097152</td>
</tr>
<tr>
<td>64</td>
<td>0.842684</td>
<td>16384</td>
<td>0.842701</td>
<td>4194304</td>
</tr>
<tr>
<td>128</td>
<td>0.842696</td>
<td>32768</td>
<td>0.842701</td>
<td>8388608</td>
</tr>
</tbody>
</table>

printf("Integral = %10.6f\n", value);

Partly because we would otherwise have to type the integrand three times and partly because we wish to make the program relatively flexible, we have defined the integrand as a user-defined function, whose definition is conveyed by the coding

```c
float func( float x ) /* Define integrand as function */
{
    return 2.0*exp(-pow(x,2))/sqrt(3.14159265);
}
```

which must be placed before the main program. A full listing of this program, named `trapezoidal.c`, is presented in Section 13.B. Simply changing the definition of the function `FUNC` will create a program that will integrate a different function.

For the sake of an explicit example, we have defined `func` in `trapezoidal.c` to return the integrand for the error function as defined in Eq. (13.26). To compile and run this program to evaluate erf(1.0) for various N, we execute the statements

```c
cc -o trapezoidal.xc trapezoidal.c -lm
./trapezoidal.xc
```

The results for various divisions of the integration interval are shown in Table 13.7. 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 (Δx is small enough) to give accurate results but not so large (Δx not so small) as to trigger problems from computer roundoff. We conclude that, to six digits, the value of this integral is 0.842701.

---

39To 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.

40Don’t overlook the final item (-lm) in the first statement. It is necessary to tell the compiler to link in the C mathematical libraries.
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).\footnote{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.}

\subsection{Using Numerical Recipes}

\textit{Numerical Recipes} provides many integration procedures cast in the form of functions, including \texttt{trapzd.c}, \texttt{qtrap.c}, \texttt{qsimp.c}, and \texttt{qromb.c}. Since it can be used by itself and it is also used in the interior of the other routines, \texttt{trapzd.c} is the workhouse of the entire suite. It is called by the statement

\begin{verbatim}
    s = trapzd( func, a, b, i );
\end{verbatim}

When invoked, \texttt{trapzd.c} evaluates the integral of the function defined by \texttt{func} over the limits from \texttt{a} to \texttt{b}, dividing that interval into $2^{i-1}$ segments\footnote{The parameter \texttt{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, \texttt{i} should probably be kept less than about 15.} and returning the result in the variable \texttt{s}. Note, however, that this procedure must be called in a sequence of calls, starting with \texttt{i = 1}, then with \texttt{i = 2}, \texttt{i = 3}, etc. The variable \texttt{s} is initialized in the first call and then incremented appropriately with each subsequent call for the next larger value of \texttt{i}. At base, \texttt{trapzd.c} 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 C driving program for using \texttt{trapzd.c} must be written by the user and will have the general form listed in Table 13.8.\footnote{A somewhat more elaborate demonstration program named \texttt{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 \textit{Local Guide}.)} This program defines the function to be integrated, establishes the desired limits, invokes \texttt{trapzd} in a properly constructed loop, and displays each value as it is obtained.

For the sake of an explicit example, we have chosen to evaluate $\text{erf}(1.0) = \frac{2}{\sqrt{2\pi}} \int_0^1 e^{-x^2} \, dx$. The steps in evaluating the desired integral with this program then are

1. Create the driving program, either by copying the demonstration program \texttt{xtrapzd.c} from the directory\footnote{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/...} \$NRHEAD/recipes_c-ansi/demo/src to the default directory, giving it the name \texttt{errtrapzd.c}, and editing it appropriately or by creating the file \texttt{errtrapzd.c} from scratch with an available text editor.

2. Copy the recipe \texttt{trapzd.c} from the directory \$NRHEAD/recipes_c-ansi/recipes and the header file \texttt{nr.h} from the directory \$NRHEAD/recipes_c-ansi/include to the default directory.

3. Compile and link the executable module with the statement
Table 13.8: The C program errtrapzd.c.

```c
/* PROGRAM errtrapzd */

#include <stdio.h> /* Load standard i/o routines */
#include <math.h> /* Load standard math functions */
#include "nr.h" /* Load necessary recipes headers */

float func(float x) /* Define integrand as function */
{
    return 2.0*exp(-pow(x,2))/sqrt(3.14159265);
}

main()
{
    float a, b, s; /* For limits, sum */
    int numit, i; /* For number of iterations */
    numit = 14; /* Set number of iterations */
    a = 0.0; b = 1.0; /* Set limits */

    /* Head table, then invoke trapzd iteratively */
    /* and display result at each step */
    printf("%6s %24s\n","n","approx. integral");
    for (i=1;i<=numit;i++){
        s=trapzd( func, a, b, i );
        printf("%6d %20.6f\n",i,s);
    }
}
```

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 13.9. 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. It is invoked with a statement of the form

```c
s = qtrap( func, a, b );
```

where func defines the function to be integrated, a and b are the lower and upper limits, and s is the variable in which the routine returns the final value. Thus, a driving program to invoke qtrap.c is simpler than one using trapzd.c directly and might have the form shown in Table 13.10. The

45In some cases, e.g., when the value of the integral being addressed is actually zero, it may be prudent to edit qtrap.c to replace the convergence criterion based on fractional change with one based on absolute change in consecutive iterates.
Table 13.9: Output from errtrapzd.c.

<table>
<thead>
<tr>
<th>n</th>
<th>approx. integral</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.771743</td>
</tr>
<tr>
<td>2</td>
<td>0.825263</td>
</tr>
<tr>
<td>3</td>
<td>0.838368</td>
</tr>
<tr>
<td>4</td>
<td>0.841619</td>
</tr>
<tr>
<td>5</td>
<td>0.842431</td>
</tr>
<tr>
<td>6</td>
<td>0.842633</td>
</tr>
<tr>
<td>7</td>
<td>0.842684</td>
</tr>
<tr>
<td>8</td>
<td>0.842697</td>
</tr>
<tr>
<td>9</td>
<td>0.842700</td>
</tr>
<tr>
<td>10</td>
<td>0.842701</td>
</tr>
<tr>
<td>11</td>
<td>0.842701</td>
</tr>
<tr>
<td>12</td>
<td>0.842701</td>
</tr>
<tr>
<td>13</td>
<td>0.842701</td>
</tr>
<tr>
<td>14</td>
<td>0.842701</td>
</tr>
</tbody>
</table>

Table 13.10: 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 */
}
```

steps in evaluating the desired integral with this program are

1. Create the driving program, either by copying the demonstration program xqtrap.c from the directory giving it the name errqtrap.c, and editing it appropriately, or by creating the file errqtrap.c from scratch with an available text editor.

2. Copy any necessary recipes and include files to the default directory. In the present case, the needed recipe trapzd.c and header file nr.h presumably remain from the first example. Each of the recipes qtrap.c and — discovered in a first (failed) attempt at compilation — nrutil.c must be copied from the directory $NRHEAD/recipes_c-ansi/recipes.
3. Compile and link the executable module with the statement

```
cc -o errqtrap.xc errqtrap.c trapzd.c qtrap.c nrutil.c -lm
```

4. Run the program with the statement 

```
./errqtrap.xc
```

The output from the program will be

```
Result using QTRAP = 0.842700
```

which is certainly in agreement with earlier results (Note that failure to converge will result in a message that includes “Too many steps ...” being printed on the screen. Here, the maximum number of steps is 20 by default, that value being defined by an editable parameter in the subroutine `qtrap.c`. After printing this message, the subroutine returns control to the operating system.)

The *Numerical Recipes* procedures `qsimp.c` and `qromb.c` are used in a way quite parallel to that of `qtrap.c`. The generation of programs to evaluate this test integral using each of these additional routines is left to the exercises. That these additional routines also yield the value 0.842701 is not at all a surprise.

Throughout this subsection, we have confined our explicit examples to integrals falling into our first category. If the integral of interest happens to depend on a parameter, our approach needs refinement. When the parameter occurs in the limits of integration, e.g., as the upper limit, we simply invoke the integrating routine in a loop, specifying a different upper limit with each pass through the loop. When the parameter occurs embedded in the integrand, we still need a loop. In addition, however, to communicate the parameter from the main program to the function, we must either (1) add that parameter as an argument to the function and edit the numerical recipes routine to accept that parameter as an argument as well or, alternatively, (2) arrange for the main program to store the parameter in a global variable (by declaring the variable to be used for the parameter ahead of and outside all other program blocks and then using the declared variable name both in the function and in the main program when values are assigned to the parameter).\(^{46}\)

### 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 \(f(t) = f_0 e^{-bt}\), where both \(b\) and \(f_0\) are positive. Let \(x(0) = x_0\) and \(v(0) = v_0\). Use symbolic integration to find \(x(t)\) and \(v(t)\) by evaluating the integrals in Eq. (13.3). Then, find and interpret both the limits of these two results as \(t \to \infty\) and the Taylor expansion of these two results for small \(t\).

\(^{46}\)See footnote 41.
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
\[ \frac{1}{2} m \left( \frac{dx}{dt} \right)^2 - G\frac{mM}{x} = -G\frac{mM}{x_0} \quad \Rightarrow \quad \frac{dx}{dt} = -\sqrt{2GM} \sqrt{\frac{1}{x} - \frac{1}{x_0}} \]
(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: Evaluate the time required for the first half of the journey, which involves a convergent but improper integral.

13.7. According to the quantum theory, the probability that the electron in the ground state of the hydrogen atom will be found between the center of the atom and some radius \( r \) is given by
\[ P(r) = \frac{4}{a^3} \int_0^r e^{-2r'/a} r'^2 \, dr' = 4 \int_0^{r/a} e^{-2\rho} \rho^2 \, d\rho \]
where \( a \) is the Bohr radius and \( \rho = r'/a \). Evaluate this integral symbolically. Then plot and comment on a graph of \( P(r) \) versus \( r/a \).

13.8. Consider a source consisting of two uniformly charged disks, each of radius \( a \) and each oriented with its center on the \( z \) axis and its plane perpendicular to the \( z \) axis. Let one disk have its center at \((0,0,ca)\) and carry a positive charge density \( \sigma \) and the other have its center at \((0,0,-ca)\) and carry a negative charge density \(-\sigma\). Using a symbolic program, show that the on-axis electrostatic potential established by this source is given by
\[ V(z) = \frac{\sigma}{2\epsilon_0} \left[ \sqrt{a^2 + (z-ca)^2} - |z-ca| - \sqrt{a^2 + (z+ca)^2} + |z+ca| \right] \]
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, \( r = z \), for which \( z > a \) by setting up and evaluating the integral

\[
V(0, 0, z) = \frac{1}{4\pi \varepsilon_0} \int_0^a \frac{\rho(r') \, dr'}{|r - r'|}
\]

symbolically to show that \( V(0, 0, z) = Q/4\pi \varepsilon_0 z \). \( (Q \) is the total charge in the distribution.\) This result demonstrates formally that the potential at a point outside a spherically symmetric charge distribution can be evaluated by regarding the charge to be concentrated at the center of the distribution. \textit{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. \textit{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

\[
I_{ij} = \int [(x_i^2 + x_j^2 + x_k^2) \delta_{ij} - x_i x_j] \, dm
\]

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

\[
x_{mn} = \int_{-\infty}^{\infty} \psi_m^*(x) x \psi_n(x) \, dx \quad \text{and} \quad p_{mn} = \int_{-\infty}^{\infty} \psi_m^*(x) \frac{h}{i} \frac{\partial}{\partial x} \psi_n(x) \, dx
\]

are important in a variety of contexts. For a particle in an infinitely deep potential well that extends over the region \( -a \leq x \leq a \),

\[
\psi_n(x) = \begin{cases} 
\frac{1}{\sqrt{a}} \cos \frac{n\pi x}{2a} & n = 1, 3, 5, \\
\frac{1}{\sqrt{a}} \sin \frac{n\pi x}{2a} & n = 2, 4, 6, \ldots
\end{cases}
\]

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

\[
x_{mn} = 16a \pi (-1)^{(m+n+1)/2} \frac{m n}{(m^2 - n^2)^2}
\]

\[
p_{mn} = \frac{2ih}{a} (-1)^{(m+n+1)/2} \frac{m n}{(m^2 - n^2)}
\]

otherwise. Note that, for purposes of translating the general integrals above to the circumstances of this exercise, the wave functions should both be regarded as zero outside of the interval \( -a \leq x \leq a \).

13.12. The sawtooth wave is defined by

\[
f(x) = \frac{x}{l}; \quad -l \leq x \leq l
\]

Use symbolic integration to find the Fourier coefficients \( a_n \) and \( b_n \) for this function and then generate graphs showing the function given by the truncated series

\[
f_{\text{trunc}}(x) = \frac{a_0}{2} + \sum_{n=1}^{N} \left( a_n \cos \frac{n\pi x}{l} + b_n \sin \frac{n\pi x}{l} \right)
\]

for various values of \( N \), including 0, 1, 2, 3, and 10.
13.13. The Legendre polynomials \( L_n(x) \), \( n = 0, 1, 2, \ldots \), are orthogonal on the interval \(-1 \leq x \leq 1\) with weight 1. In particular,
\[
\int_{-1}^{1} L_m(x) L_n(x) \, dx = \frac{2}{2n + 1} \delta_{mn}
\]
Any function defined over the interval \(-1 \leq x \leq 1\) can then be expanded in the Legendre series
\[
f(x) = \sum_{n=0}^{\infty} c_n L_n(x)
\]
(a) Show by hand that the coefficient \( c_n \) in this expansion is given by
\[
c_n = \frac{2n + 1}{2} \int_{-1}^{1} f(x) L_n(x) \, dx
\]
(b) Use symbolic integration to find \( c_n \) for \( n = 0, 1, 2, 3, 4, 5 \) and 6 in the Legendre expansion for the function
\[
f(x) = \begin{cases} 
-1 & -1 < x < 0 \\
1 & 0 < x < 1
\end{cases}
\]
(c) Graph the functions defined by the partial sums \( \sum_{n=0}^{N} c_n L_n(x) \) for \( N = 0, 1, 2, 3, 4, 5, \) and 6.

Hint: Quite possibly the symbolic program you are using has the Legendre polynomials built in somehow, and you should study its manuals to find out how to invoke them. Just in case that isn’t true, the first nine Legendre polynomials are
\[
\begin{align*}
L_0(x) &= 1 \\
L_1(x) &= x \\
L_2(x) &= \frac{1}{2} (3x^2 - 1) \\
L_3(x) &= \frac{1}{2} (5x^3 - 3x) \\
L_4(x) &= \frac{1}{2} (35x^4 - 30x^2 + 3) \\
L_5(x) &= \frac{1}{2} (63x^5 - 70x^3 + 15x) \\
L_6(x) &= \frac{1}{2} (231x^6 - 315x^4 + 105x^2 - 5) \\
L_7(x) &= \frac{1}{2} (429x^7 - 693x^5 + 315x^3 - 35x) \\
L_8(x) &= \frac{1}{2} (12155x^8 - 25740x^6 + 18018x^4 - 4620x^2 + 35)
\end{align*}
\]

13.15.2 . . using Numerical Methods

13.14. To deduce Simpson’s rule, we start by supposing three consecutive values \( f_1, f_2, \) and \( f_3 \) of the integrand, where for simplicity in notation we take the points of evaluation to be \( x_1 = x_2 - \Delta x, x_2, \) and \( x_3 = x_2 + \Delta x. \) Using a symbol manipulating program to do the algebra and calculus, (a) find the coefficients \( A, B, \) and \( C \) needed to make the parabola \( Ax^2 + Bx + C \) pass through the three points \( (x_i, f_i), i = 1, 2, 3, \) (b) integrate that parabola over the interval \( x_1 < x < x_3 \) to find that
\[
\int_{x_1}^{x_3} f(x) \, dx \approx \int_{x_1}^{x_3} (Ax^2 + Bx + C) \, dx = \frac{\Delta x}{3} \left( f_1 + 4f_2 + f_3 \right)
\]
(c) show that this result actually gives the correct value for \( f(x) = x^3 \) and, finally, (d) deduce the (extended) Simpson’s rule of Eq. (13.63). 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.

13.17. (a) Deduce the three-point Gaussian formula, for which—parallelining 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 \( L_3(x) = (5x^3 - 3x)/2. \)
13.18. The (normalized) wave functions for a quantum harmonic oscillator in its first and second excited states \((n = 1, n = 2)\) are

\[
\psi_1(x) = \sqrt{2} \left( \frac{m\omega}{\pi\hbar} \right)^{1/4} e^{-y^2/2}; \quad \psi_2(x) = \frac{1}{\sqrt{2}} \left( \frac{m\omega}{\pi\hbar} \right)^{1/4} (2y^2 - 1) e^{-y^2/2}
\]

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

\[
f(v) = 4\pi \left( \frac{m}{2\pi kT} \right)^{3/2} \int_0^\infty e^{-mv^2/2kT} v^2 dv'
\]

for the fraction of the molecules having speed less than \(v\). Using numerical means, explore this integral as a function of \(v\). \textit{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. \textit{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 \(8\pi\hbar^3/\alpha^3\) as a function of \(T\) in units of \(\hbar\nu_0/k\) 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

\[
P(t) = \frac{1}{\sqrt{2\pi} \sigma} e^{-t^2/(2\sigma^2)}
\]

Thus, the probability of finding a value between \(a\) and \(b\) is given by

\[
P(a, b) = \frac{1}{\sqrt{2\pi} \sigma} \int_a^b e^{-t^2/(2\sigma^2)} dt
\]

Show analytically that \(P(-x, x) = \text{erf}(x/\sqrt{2\sigma})\), 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? \textit{Hint:} Since the gravitational potential is \(-GmM/x\), conservation of energy yields

\[
\frac{1}{2}m \left( \frac{dx}{dt} \right)^2 - G\frac{mM}{x} = -G\frac{mM}{x_0} \quad \Rightarrow \quad \frac{dx}{dt} = -\sqrt{2GM} \sqrt{\frac{1}{x} - \frac{1}{x_0}}
\]

(The negative square root is taken because \(x\), the distance to the sun, is known to be decreasing.) This expression then leads to the value

\[
T_{\text{midhalf}} = \frac{1}{\sqrt{2GM}} \int_{x_0/4}^{3x_0/4} \left( \frac{1}{x} - \frac{1}{x_0} \right)^{-1/2} dx
\]

\textit{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—involve 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

\[
P(r) = \frac{4}{a^3} \int_0^r e^{-2\alpha/a} r^2 \, dr = 4 \int_0^{r/a} e^{-2\rho^2} \, d\rho
\]

where \( a \) is the Bohr radius and \( \rho = r/a \). Using numerical integration, evaluate this integral as a function of its upper limit. Then plot and comment on a graph of \( P(r) \) versus \( r/a \).

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

\[
\omega t = \int_0^{\theta} \frac{d\phi}{(1 - k^2 \sin^2 \phi)^{1/2}}
\]

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

\[
J_n(x) = \frac{1}{\pi} \int_0^{\pi} \cos(n\theta - x \sin \theta) \, d\theta
\]

By evaluating this integral numerically as a function of \( x \) for different values of \( n \), obtain graphs of \( J_0(x) \), \( J_1(x) \), and \( J_2(x) \) over the range \( 0 \leq x \leq 10 \).

13.28. The Bessel function \( J_1(x) \) can be defined by the integral

\[
\frac{1}{x} J_1(x) = \frac{2}{\pi} \int_0^1 (1 - u^2)^{1/2} \cos(xu) \, du
\]

Using this definition, obtain a graph of \( J_1(x) \) versus \( x \) over the range \( 0 \leq x \leq 10 \).

13.29. A circular ring of radius \( a \) resides in the \( xy \) plane with its center at the origin and carries a charge \( Q \) uniformly distributed about its perimeter. The electrostatic potential established by this ring at an observation point whose cylindrical coordinates are \( (r, \phi, z) \) is

\[
V(r, \phi, z) = \frac{Q}{4\pi\epsilon_0 a} \int_0^{\pi} \left( 1 - \frac{2r}{a} \cos \phi' + \frac{r^2 + z^2}{a^2} \right)^{-1/2} \, d\phi'
\]
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

$$
\mathbf{B}(x, z) = \frac{a^2}{\mu_0 I'/2\pi a} \int_{0}^{\infty} \frac{z \cos \phi' \hat{i} + (a - x \cos \phi') \hat{k}}{|x^2 + z^2 + a^2 - 2ax \cos \phi'|^{3/2}} \, d\phi'
$$

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

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 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

\[ f(v) = \int_0^v \frac{r^2}{v} e^{-v^2/2} \, dv \]

(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 \( \text{erf}(1.0) \). 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
13.15. EXERCISES

request a new set of limits. Include a smooth way (other than simply CONTROL-C) to terminate execution of the program. Be sure to test your program.

13.46. Edit the driving program errqtrap.c (or—better—the driving program errqsimp.c created in an earlier exercise) so that it evaluates the integral

\[ F(x) = \int_0^x g(s) \, ds \]

as a function of the upper limit \( x \), printing a table of values of \( x \) and \( F(x) \) for values of \( x \) ranging in steps \( \Delta x \) from \( x = a \) to \( x = b \), where \( g(s) \) is defined by a function subprogram and \( \Delta x \), \( a \), and \( b \) are to be entered at execution time. Use the error function \( \text{erf}(x) \) as defined in Eq. (13.26) as a test integrand for your program.

13.47. Use trapzd.c, qtrap.c, qsimp.c, and qromb.c in a succession of programs to evaluate the quantity

\[ f(v) = \int_0^v v^2 e^{-v^2/2} \, dv \]

(a) for \( v = 1.0 \) and (b) as a function of \( v \) over the range \( 0.0 \leq v \leq 3.0 \). Cast your response to part (b) so that it writes the values of \( v \) and \( f(v) \) into a (text) file that could then be imported into another program for graphical display. Statements for opening, writing to, and closing a file in C are introduced in Chapter 9.

13.48. Use Numerical Recipes in C to address Exercise 13.19, which involves evaluating an integral as a function of its upper limit.

13.49. Use Numerical Recipes in C to address Exercise 13.27, which involves evaluating an integral as a function of a parameter in the integrand. You will, of course, have to refresh your understanding of global variables in C.
13.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>  
#include <math.h>  

float func( 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( "
Lower 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

$$f(x) = 0$$

(14.1)

If $f(x)$ is simple, we may be able to find a closed form, analytic solution. More often, however, $f(x)$ is sufficiently complicated that approximate, numerical methods are needed. Further, many functions will have several roots, some of which may be physically meaningless. Thus, we must learn not only how to find the roots but also how to sort the physically meaningful roots from a possibly larger number of mathematically acceptable ones, the “extras” of which are said to be spurious. We begin this chapter by identifying several physical situations, the full addressing of which requires finding one or more roots of some function. Then we illustrate how to use symbolic algebra systems to approach those that can be addressed analytically, describe a few of many available numerical algorithms, and describe ways to find roots using a variety of numerical approaches and computational tools. Briefly at the end, we comment about the more complicated issue of finding roots of sets of simultaneous linear and non-linear equations. Until that point, our discussion will focus on functions of a single variable.

Whatever the function and whatever the approach, the first step in seeking roots should always be to learn as much as possible about the nature of the function and its roots. Further, since numerical methods in particular—most of them iterative—require a starting guess or guesses and will converge more or less rapidly and reliably depending on the quality of those guesses, a priori knowledge of the approximate location of roots is essential. Thus, we should always start by drawing a graph of $f(x)$ in sufficient detail to reveal the approximate location of the roots of interest. Since the focus of this chapter is not graphing (and graphing has been fully addressed in earlier chapters), we shall carry out this step once in Section 14.1 as we present several sample problems rather than carrying it out repeatedly in later sections.\(^1\)

14.1 Sample Problems

In this section, we identify several physical contexts in which the essential computational problem is to find the roots of some function, and we obtain graphs of the appropriate functions for later reference.

\(^1\)The graphs could, of course, be produced in any number of ways. Except for Fig. 14.2 (which was produced with \texttt{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

\[
V(x) = \frac{x^3}{10000} + \frac{x^2}{200} - \frac{x}{500} - \frac{1}{2} \quad (14.3)
\]

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

\[
F = 0 \quad \Rightarrow \quad \frac{dV}{dx} = 0 \quad (14.5)
\]

or where the graph of \( V(x) \) has a horizontal tangent. Further, evaluated at a point of equilibrium, \( d^2 V/dx^2 > 0 \) implies that the equilibrium is stable while \( d^2 V/dx^2 < 0 \) 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 \( d^2 V/dx^2 \).

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

\[
-35.00 < x_4 < -30.00 \quad \text{and} \quad -2.50 < x_5 < 2.50 \quad (14.7)
\]

This function is, of course, not particularly realistic as a potential energy. We can, however, provide a physical context for at least a portion of \( V(x) \). If we confine our attention to the region around the one minimum it possesses, we can interpret the function as the potential energy of an anharmonic oscillator, for which—if that minimum occurs at \( x = 0 \)—we might write \( V(x) = kx^2/2 + ax^3 \) while imposing the constraint that \( a \), the coefficient of the cubic perturbation from the potential energy of a simple harmonic oscillator, be small. The expression in Eq. (14.2) simply places the minimum at a different value of \( x \) and adds a constant to the potential energy. In what follows, we will explore this function over a wider range of values of \( x \) than is physically meaningful. The pedagogic advantage of Eq. (14.2) is that it combines many of the important features of potential energy functions with, as we shall see, tractability by a variety of different approaches.
14.1. SAMPLE PROBLEMS

Figure 14.1: Graph of the potential energy of Eq. (14.3).

Figure 14.2: A system of two masses and three springs. The surface on which the two objects rest is assumed to be frictionless.

14.1.3 Natural Frequencies of Oscillating Systems

Suppose we seek the natural frequencies of oscillation for the system shown in Fig. 14.2, which consists of two objects, each having mass \( m \). Let these objects move in one dimension on a horizontal, frictionless surface, let them be connected to one another with a spring having constant \( k' \), and let each be connected to the nearer wall with a spring having constant \( k \). Further, let the position of each be measured from its equilibrium position. Then, Newton’s second law combined with Hooke’s law leads to the equations of motion\(^3\)

\[
m \frac{d^2 x_1}{dt^2} = -kx_1 + k'(x_2 - x_1) \quad \text{and} \quad m \frac{d^2 x_2}{dt^2} = -kx_2 - k'(x_2 - x_1)
\]  
(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 \( \bar{\kappa} = k'/k \), and introduce \( \bar{t} = \omega_0 t \) to find that

\[
\frac{d^2 \bar{x}_1}{d\bar{t}^2} = -(1 + \bar{\kappa})\bar{x}_1 + \bar{\kappa}\bar{x}_2 \quad \text{and} \quad \frac{d^2 \bar{x}_2}{d\bar{t}^2} = \bar{\kappa}\bar{x}_1 - (1 + \bar{\kappa})\bar{x}_2
\]  
(14.9)

Next, seeking sinusoidal (or simple harmonic) solutions, we suppose that\(^3\)

\[
\bar{x}_1(\bar{t}) = \bar{x}_{10} \cos \omega_0 \bar{t} \quad \text{and} \quad \bar{x}_2(\bar{t}) = \bar{x}_{20} \cos \omega_0 \bar{t}
\]  
(14.10)

\(^3\)These equations were also discussed in Section 11.1.6.
CHAPTER 14. FINDING ROOTS

Figure 14.3: The polynomial \( D(\omega) \) in Eq. (14.12) for (a) \( \pi = 0.25 \), (b) \( \pi = 0.5 \), (c) \( \pi = 1.0 \), and (d) \( \pi = 2.0 \).

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

\[
\begin{pmatrix}
1 + \pi - \omega^2 & -\pi \\
-\pi & 1 + \pi - \omega^2
\end{pmatrix}
\begin{pmatrix}
\pi_{10} \\
\pi_{20}
\end{pmatrix} = 0 \tag{14.11}
\]

The solution of this equation for the unknowns \( \pi_{10} \) and \( \pi_{20} \) will be trivial (i.e., both zero—a particularly uninteresting motion) unless the determinant of the matrix of coefficients happens itself to be zero, i.e., unless the characteristic equation

\[
D(\omega) = \begin{vmatrix}
1 + \pi - \omega^2 & -\pi \\
-\pi & 1 + \pi - \omega^2
\end{vmatrix} = (1 + \pi - \omega^2)^2 - \pi^2 = \omega^4 - 2\omega^2(1 + \pi) + 1 + 2\pi = 0 \tag{14.12}
\]

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 \( \pi = k'/k \), 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 \( \pi \) 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 \( \pi \), while the upper of the two roots increases steadily as \( \pi \) increases.

14.1.4 Range of Projectile in Viscous Medium

For a fourth example, suppose we seek the range of a projectile of mass \( m \) fired with the initial speed \( v_0 \) at angle \( \theta \) up from the horizontal in a viscous medium having damping constant \( b \). Supposing the
projectile to move in the $xy$ plane, with $x$ horizontal and $y$ vertical, we begin by invoking Newton’s second law\(^4\) to write the equations of motion

\[
\frac{md^2x}{dt^2} = -b\frac{dx}{dt} \quad \text{and} \quad \frac{md^2y}{dt^2} = -mg - b\frac{dy}{dt}
\] (14.13)

Here, $g$ is the acceleration of gravity (which we take to be a positive number). These equations are to be solved subject to the initial values

\[
x(0) = 0 \quad \text{,} \quad v_x(0) = v_0 \cos \theta \quad \text{,} \quad y(0) = 0 \quad \text{and} \quad v_y(0) = v_0 \sin \theta
\] (14.14)

The solution to this problem—see Chapter 11 on ordinary differential equations—is

\[
x(t) = \frac{mv_0 \cos \theta}{b} \left(1 - e^{-bt/m}\right)
\] (14.15)

\[
y(t) = -\frac{mgt}{b} + \frac{m}{b} \left(v_0 \sin \theta + \frac{mg}{b}\right) \left(1 - e^{-bt/m}\right)
\] (14.16)

Finally, to find the range $R$, we seek the value of $x$ at that non-zero value of $t$ for which $y(t) = 0$. Thus, we need to find the non-zero solution, say $t_1$, of the equation

\[-\frac{mgt}{b} + \frac{m}{b} \left(v_0 \sin \theta + \frac{mg}{b}\right) \left(1 - e^{-bt/m}\right) = 0
\] (14.17)

and then evaluate $x(t_1)$. Equivalently, if we introduce the dimensionless time $\tau = bt/m$ and the dimensionless parameter $\alpha = bv_0/(mg)$, the equation whose root we seek becomes

\[f(\tau) = \tau - \left(1 + \alpha \sin \theta\right) \left(1 - e^{-\tau}\right) = 0
\] (14.18)

Once we have found the desired root, say $\tau_1$, we then determine the range $R$ by substituting $\tau_1$ into Eq. (14.15), finding that

\[
\frac{R(\theta, \alpha)}{v_0^2/g} = \cos \frac{\theta}{\alpha(1 + \alpha \sin \theta) \tau_1}
\] (14.19)

The mathematical task confronting us involves finding the non-zero root of the function $f(\tau)$ defined in Eq. (14.18). More specifically, finding the angle $\theta$ at which the projectile should be fired to achieve maximum range would entail

- Choosing a value of $\alpha = bv_0/(mg)$.
- Finding the non-zero root of $f(\tau)$ for several values of $\theta$ ranging from 0 to $\pi/2$.
- Calculating and plotting values of $R(\theta, \alpha)$.
- Finding the value of $\theta$ corresponding to the peak in a graph of $R(\theta, \alpha)$ versus $\theta$.

Note that $\alpha$ increases as $b$ (the damping) increases and as $v_0$ increases but decreases as $m$ increases.

To gain some insight into the nature of the desired roots, we begin by plotting the family of graphs of $f(\tau)$ versus $\tau$ for various values of $\theta$ with fixed $\alpha$. After some exploration using techniques introduced in earlier chapters, we arrive at the several graphs shown in Fig. 14.4. Every graph exhibits a root at $\tau = 0$, corresponding to the moment of launch. Though it is hard to judge, each graph for $\theta = 0^\circ$ also exhibits a second root at $\tau = 0$, since a projectile launched at $\theta = 0^\circ$ returns to its initial altitude immediately. As the angle of launch is increased in each case, however, the second root moves to larger and larger values of $\tau$. Our task in subsequent sections will be to find numerical values for that second root for various values of $\alpha$ and $\theta$, determine the range for each, and find the particular angle at which the range is greatest for a given $\alpha$.

\(^4\)Compare Eqs. (11.3) and (11.4).
Figure 14.4: Graphs of $f(\tau)$ versus $\tau$. The individual frames in this display correspond to different values of $\alpha$; the graphs in each frame correspond to different values of $\theta$, ranging in $15^\circ$ increments from $0^\circ$ for the highest graph to $90^\circ$ for the lowest graph in each frame. The scalings have been chosen to reveal the nature of the roots most clearly and are different in each frame.

As a quick aside, we can reassure ourselves that this approach is appropriate by examining the limits of Eqs. (14.18) and (14.19) as $b$ becomes small. In that limit, both $\tau = bt/m$ and $\alpha = bv_0/(mg)$ also become small. To assess the limit, we need to expand Eq. (14.18) to second order in $\tau$ and $\alpha$, finding that

$$0 = \tau - (1 + \alpha \sin \theta) \left[ 1 - \left( 1 - \tau + \frac{\tau^2}{2} \right) \right] = \tau + (1 + \alpha \sin \theta) \left[ -\tau + \frac{\tau^2}{2} \right]$$

$$= -\alpha \tau \sin \theta + \frac{\tau^2}{2} = 0 \quad \Rightarrow \quad \tau = 2\alpha \sin \theta$$

(14.20)

Then, substituting this result into Eq. (14.19), we find that

$$\frac{R(\theta, \alpha \to 0)}{v_0^2/g} = \frac{\cos \theta}{\alpha(1 + \alpha \sin \theta)} (2\alpha \sin \theta) = \frac{2\sin \theta \cos \theta}{(1 + \alpha \sin \theta)} \to 2 \sin \theta \cos \theta$$

(14.21)

which is in complete agreement with the known result for the range of a projectile in the absence of air resistance.
14.1.5 Energy Levels in a Quantum Well

The quantum mechanical analysis for the energy levels of a particle of mass $m$ in a one-dimensional quantum well characterized by the potential energy

$$V(x) = \begin{cases} \infty & x < 0 \\ -V_0 & 0 \leq x \leq a \\ 0 & a \leq x \end{cases} \quad (14.22)$$

leads to the conclusion that the allowed energies $E$ should satisfy the equation\(^5\)

$$s \cot s = -\sqrt{c^2 - s^2} \quad (14.23)$$

where $c^2 = 2ma^2V_0/h^2$ and $s^2 = c^2(1 - E/V_0)$ or $E = -V_0(1 - s^2/c^2)$. To find the energies, we must thus solve Eq. (14.23) for acceptable values of $s$ once the depth of the well conveyed by the (fixed) value of $c$ has been specified.

A simpler version of this equation emerges if we square it (thereby possibly introducing spurious roots because the squared equation is also consistent with a plus sign on the right hand side) and then invoke the trigonometric identity $1 + \cot^2 \theta = \sin^{-2} \theta$ to find that

$$s^2 \cot^2 s = s^2 (\sin^{-2} s - 1) = c^2 - s^2 \quad \Rightarrow \quad \sin s = \pm \frac{s}{c} \quad (14.24)$$

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

$$\sin s = \pm 0.04s \quad \text{or} \quad \sin s \mp 0.04s = 0 \quad (14.25)$$

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 nine 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

$$\begin{array}{ll}
\text{root 1u} & -0.5 < s < 0.5 \\
\text{root 2u} & 2.5 < s < 3.5 \\
\text{root 3u} & 6.0 < s < 7.0 \\
\text{root 4u} & 8.5 < s < 9.5 \\
\end{array} \quad \begin{array}{ll}
\text{root 5u} & 12.5 < s < 13.5 \\
\text{root 6u} & 14.5 < s < 15.5 \\
\text{root 7u} & 19.0 < s < 20.0 \\
\text{root 8u} & 20.5 < s < 21.5 \\
\end{array} \quad (14.26)$$

and that, for the lower sign, the nine roots are bounded—again crudely—by

$$\begin{array}{ll}
\text{root 1l} & -0.5 < s < 0.5 \\
\text{root 2l} & 3.0 < s < 4.0 \\
\text{root 3l} & 5.5 < s < 6.5 \\
\text{root 4l} & 9.5 < s < 10.5 \\
\text{root 5l} & 11.5 < s < 12.5 \\
\end{array} \quad \begin{array}{ll}
\text{root 6l} & 16.0 < s < 17.0 \\
\text{root 7l} & 17.5 < s < 18.5 \\
\text{root 8l} & 22.5 < s < 23.5 \\
\text{root 9l} & 23.5 < s < 24.5 \\
\end{array} \quad (14.27)$$

\(^5\)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

$$ f(x) = ax + b = 0 \implies x = -\frac{b}{a} \quad (14.28) $$

and that the quadratic formula

$$ f(x) = ax^2 + bx + c = 0 \implies x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} = \frac{2c}{-b \mp \sqrt{b^2 - 4ac}} \quad (14.29) $$

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

$$ f(x) = \sin(kx) = 0 \implies x = \frac{n\pi}{k}, \quad n = 0, \pm 1, \pm 2, \ldots \quad (14.30) $$
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

\[ \frac{a + x}{b + x^2} - c = 0 \quad \implies \quad a + x - c(b + x^2) = -cx^2 + x + a - cb = 0 \]  

(14.32)

converts the problem into one involving a quadratic polynomial. Beyond these few cases, very few roots can be found symbolically.

### 14.5 Finding Roots Symbolically with Mathematica

The Mathematica command for solving a wide variety of equations is `Solve`. It takes two mandatory arguments, the first giving the equation to be solved and the second specifying the variable for which a solution is desired. Thus, the statement

\[ \text{In[1]:= Solve[ a}\times x + b == 0, x ] \]

\[ \text{Out[1]= \{ \{ x \rightarrow -\frac{b}{a} \} \} } \]

returns the solution \( x = -b/a \) in the form of a transformation rule and the statement

\[ \text{In[2]:= Solve[ a}\times x^2 + b\times x + c == 0, x ] \]

\[ \text{Out[2]= \{ \{ x \rightarrow -\frac{b - \sqrt{b^2 - 4ac}}{2a} \}, \{ x \rightarrow -\frac{b + \sqrt{b^2 - 4ac}}{2a} \} \} } \]

returns a list containing two solutions, each expressed as a transformation rule. 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 `Root[...]`, e.g.,

\[ \text{In[3]:= Solve[ a}\times x^6 - b\times x^5 + c == 0, x ] \]

\[ \text{Out[3]= \{ \{ x \rightarrow \text{Root}[c - b \#^5 + a \#^6 & , 1] \}, \{ x \rightarrow \text{Root}[c - b \#^5 + a \#^6 & , 2] \}, \ldots \} } \]

(Here, \# is a Mathematica-generated generic variable, `Root[f,n]` stands for the \( n \)-th root of the function \( f \), and the ampersand identifies what precedes it as a “pure” function.\(^6\)) Basically, this response constitutes an admission by Mathematica that it can’t solve the specified equation symbolically. Fuller description of the command `Solve`, of a third (optional) argument that specifies variables to be eliminated, and of a few options that affect the behavior of the command are described in the Mathematica manuals. Further, Mathematica will display the options and their current values in response to the statement `Options[Solve].`

\(^6\)See Section 2.2.5 in The Mathematica Book (Fourth Edition).
14.5.1 Classical Turning Points

The turning points for the potential energy in Eq. (14.3) are readily found symbolically with the statements

\[
\text{In[4]:= } V = \frac{x^3}{10000} + \frac{x^2}{200} - \frac{x}{500} - \frac{1}{2}; \\
\text{In[5]:= } \text{soln} = \text{Solve}[V == 0, x] \\
\text{Out}[5]= \left\{ \{x \rightarrow \text{mess}\}, \{x \rightarrow \text{mess}\}, \{x \rightarrow \text{mess}\} \right\}
\]

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

\[
\text{In[6]:= } \text{soln} \ // \ \text{N} \\
\text{Out}[6]= \left\{ \{x \rightarrow 9.3487 + 0. \times 10^{-20} i\}, \{-48.2683 - 8.88178 \times 10^{-16} i\}, \{-11.0804 + 0. i\} \right\}
\]

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 \(N\) is on the order of 10^{-15}. Indeed, we can support this understanding of the imaginary part by evaluating the roots to more decimal places with the statements

\[
\text{In[7]:= } \text{N[ soln, 20]} \\
\text{Out[7]}= \left\{ \{x \rightarrow 9.3487041976000522706 + 0. \times 10^{-20} i\}, \{-48.268267566751634131 + 0. \times 10^{-19} i\}, \{-11.080436630848481393 + 0. \times 10^{-20} i\} \right\}
\]

Even at this precision, the assertion that the imaginary parts are all zero is not unreasonable, given that the precision of the command \(N\) is on the order of one in the twentieth digit when its second argument is 20.\(^7\)

14.5.2 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

---

\(^7\)While different machines should give the same results for most of the digits in the real parts of these roots, the values of the imaginary parts may well be quite different from machine to machine, since those values are largely a consequence of internal round off.
14.5. FINDING ROOTS SYMBOLICALLY WITH MATHEMATICA

In[8]:= dVdx = D[ V, x ];
In[9]:= Solve[ dVdx == 0, x ]
Out[9]= 
{ 
{x -> \( \frac{2}{3} \left( -25 - 8\sqrt{10} \right) \)},
{x -> \( \frac{2}{3} \left( -25 + 8\sqrt{10} \right) \)}
}

In[10]:= soln = N[ % ]
Out[10]= 
{ 
{x -> -33.5321},
{x -> 0.198814}
}

finding first \( \frac{dV}{dx} \) and then finding the values of \( x \) at which \( \frac{dV}{dx} = 0 \). 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

In[11]:= d2Vdx2 = D[ dVdx, x ];
In[12]:= d2Vdx2 /. soln
Out[12]= 
{ -0.0101193, 0.0101193 }
In[13]:= Quit[]

concluding that the first root \( x = -33.5321 \) locates an unstable equilibrium (\( \frac{d^2V}{dx^2} < 0 \); local maximum in the potential energy) while the second root \( x = 0.198814 \) locates a stable equilibrium (\( \frac{d^2V}{dx^2} > 0 \); local minimum in the potential energy).

14.5.3 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

In[1]:= \( d = \Omega^4 - 2*\Omega^2*(1+\Kappa) + 1 + 2*\Kappa \);
In[2]:= \( d \) /. \( \Omega \to \) Sqrt[\( \Lambda \)];
In[3]:= Solve[ \% == 0, \( \Lambda \) ]
Out[3]= 
{ \{ \lambda \to 1 \}, \{ \lambda \to 1 + 2\kappa \} }
In[4]:= freqs = Sqrt[ \( \Lambda \) ] /. \%
Out[4]= 
{ 1, \sqrt{1+2\kappa} }

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

\footnote{At this point, for simplicity, we drop the overbar on \( \kappa \).}

\footnote{Evidently, we cannot simply place the list \texttt{freqs} as the first argument to \texttt{Plot}. The first argument can be a list, but the list must be constructed within the statement rather than outside of the statement.}
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 $\omega_0 = \sqrt{k/m}$.

In[5]:= Plot[ {freqs[[1]], freqs[[2]]}, { \[Kappa], 0.0, 4.0 }, PlotRange -> { 0.0, 3.0}, PlotStyle -> Thickness[0.005], AxesLabel -> { \[Kappa], \[Omega] } ]
In[6]:= Quit[]

The resulting graph is shown in Fig. 14.6.

14.5.4 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 Mathematica. We begin by defining the trajectory as given by Eqs. (14.15) and (14.16) with the statements\(^\text{10}\)

In[1]:= x = (m*v0*Cos[\[Theta]]/b) * ( 1 - Exp[-b*t/m] );
In[2]:= 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 Mathematica statements

\(^{10}\)To facilitate visualization, some of the output in the following dialog with Mathematica has been recast from the actual form presented by Mathematica.
In[3]:= \text{xb} = \text{Series}[x, \{b, 0, 1\}]

Out[3]= tv_0 \cos[\theta] - \frac{t^2 v_0 \cos[\theta] b}{2m} + O[b]^2

In[4]:= \text{yb} = \text{Simplify}[\text{Series}[y, \{b, 0, 1\}]]

Out[4]= -\frac{gt^2}{2} + tv_0 \sin[\theta] + \frac{t^2(g - 3v_0 \sin[\theta] b)}{6m} + 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

In[5]:= \text{yb} = \text{Normal}[yb];

In[6]:= \text{yb} = \text{Expand}[yb/t];

In[7]:= \text{sln} = \text{Solve}[yb == 0, t]

Out[7]= \{\{t \rightarrow \text{mess}\}, \{t \rightarrow \text{mess}\}\}

To sort out which of these two roots is the physically significant one, we evaluate the limit of this expression as $b \rightarrow 0$ with the statements

In[8]:= \text{Series}[t /. \text{sln[[1]]}, \{b, 0, 1\}]; \quad \text{Find limit of first solution.}

In[9]:= \text{Simplify}[\%, \{m > 0, g > 0\}]

Out[9]= \frac{2v_0 \sin[\theta]}{g} - \frac{2v_0^2 \sin[\theta]^2 b}{3g^2 m} + O[b]^2

In[10]:= \text{Series}[t /. \text{sln[[2]]}, \{b, 0, 1\}]; \quad \text{Find limit of second solution.}

In[11]:= \text{Simplify}[\%, \{m > 0, g > 0\}]

Out[11]= \frac{3m}{b} + \frac{v_0 \sin[\theta]}{g} + \frac{2v_0^2 \sin[\theta]^2 b}{3g^2 m} + O[b]^2

Since the second root becomes arbitrarily large as $b$ becomes small, we must reject it on physical grounds. Only the first root, which we can display more fully with the statement

In[12]:= \text{sln1} = t /. \text{sln[[1]]};

In[13]:= \text{sln2} = \text{Expand}[\text{Simplify}[\text{sln1}]]

Out[13]= \frac{3m}{2b} + \frac{3v_0 \sin[\theta]}{2g} - \frac{m \sqrt{9g^2 - 6bgv_0 \sin[\theta]/m + 9b^2v_0^2 \sin[\theta]^2/m^2}}{2bg}

is of interest. Finally, for consistency with approximations already made, we find this root to first order in $b$ with the statement

In[14]:= \text{Series}[\text{sln2}, \{b, 0, 1\}]

In[15]:= \text{tim} = \text{Simplify}[\%, \{m > 0, g > 0\}]

Out[15]= \frac{2v_0 \sin[\theta]}{g} - \frac{2v_0^2 \sin[\theta]^2 b}{3g^2 m} + O[b]^2

and find the range by substituting this result into the first order expression for the horizontal coordinate with the statements
CHAPTER 14. FINDING ROOTS

In[16]:= rang = Normal[xb] /. t -> Normal[tim];
In[17]:= rang = Series[rang, {b, 0, 1}];
In[18]:= rang = Factor[Normal[rang]]
Out[18] := 2
(We have rearranged this expression a bit from the one presented by Mathematica.) 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
\[
x_{\text{mid}} = \frac{1}{2} (x_{\text{min}} + x_{\text{max}}) \quad \text{and} \quad f_{\text{mid}} = f(x_{\text{mid}})
\]  
(14.33)

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 \(x_{\text{mid}} < x < x_{\text{min}}\) (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. ALGORITHMS FOR FINDING ROOTS NUMERICALLY

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.

14.6.2 Newton’s Method

A second—more efficient but also less stable—algorithm requires a single starting value rather than a pair of values that bracket a root. Suppose that $x_n$ is the current estimate of the position of the desired root, and let $f_n = f(x_n)$. In Newton’s method, we calculate the next iterate $x_{n+1}$ by

1. calculating the derivative of the function at $x_n$, i.e., $df(x)/dx|_{x_n} = f'_n$.
2. extrapolating the tangent line to the graph of the function—a line whose slope is $f'_n$—to the point at which it intersects the horizontal axis, i.e., finding $x_{n+1}$ by requiring it to satisfy

$$
\frac{\Delta f}{\Delta x} = \frac{0 - f_n}{x_{n+1} - x_n} = f'_n \quad \Rightarrow \quad x_{n+1} = x_n - \frac{f_n}{f'_n}
$$

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
CHAPTER 14. FINDING ROOTS

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

$$f(x) = \prod_{i=1}^{n} (x - x_i)$$  \hspace{1cm} (14.35)

(2) finds a couple of relationships satisfied by polynomials in this form, (3) assumes with only weak justification a priori that the root sought is isolated from all the others (which are clustered together), and (4) deduces that the next iterate $x_{n+1}$ for the root sought should be determined from the current guess $x_n$ by

$$x_{n+1} = x_n - \frac{N}{G \pm \sqrt{(N - 1)(NH - G^2)}}$$  \hspace{1cm} (14.36)

where the sign in the denominator is chosen so that the denominator has the larger of the two possible absolute values, $N$ is the order of the polynomial (equal to the number of roots), and $G$ and $H$ are defined by $G = f'/f$ 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.\footnote{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.} 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 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.

### 14.10 Finding Roots Numerically with PYTHON

*Note:* All PYTHON program (.py) files referred to in this chapter are available in the directory `$HEAD/python`, where (as defined in the *Local Guide*) `$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 PYTHON’s default search path. If so, the files can be found by PYTHON 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.

The PYTHON module `numpy` contains a command for finding the roots of a polynomial and the module `scipy.optimize` contains numerous routines for optimizing functions, finding minima of functions, and for finding roots of single functions and systems of functions. To simplify illustrating statements invoking these functions throughout this section, we assume that the modules have been imported with the statement

```python
import numpy as np
import matplotlib.pyplot as plt
import scipy.optimize as so
```

and we will not repeat these statements. Commands for finding roots of a function of a single variable include

• `np.roots`, which finds all roots of a polynomial if given the coefficients in the polynomial. It is invoked with a statement like

```
q = np.roots(coeffs)
```

where `coeffs` is a list that provides the coefficients of the powers of the variable in the polynomial, with the highest power as `p[0]`. This command admits no keywords.

• `so.bisect`, which is invoked with a statement like

```
q = so.bisect(func, xlb, xub)
```

where `func` is a string giving the name of a function py-file that returns the value of the function whose roots are sought and `xlb` and `xub` give the lower and upper bounds on an interval containing the desired root. The command also admits a number of keywords (with defaults), including

- `args=()`, which provides a tuple containing values to be passed as arguments to `func`.
- `xtol=2e-12` and `rtol=8.88178419...e-16`, which specify the target absolute and relative tolerances in the returned root. The solver strives to make the difference between the exact root and the returned estimate be less than `xtol + rtol * q`.
- `maxiter=100`, which provides an upper limit on the number of iterations before a message of non-convergence will be displayed.

As always with the method of bisection, the function whose root is sought must have opposite signs at the ends of the range specified by `xlb` and `xub`. Further, the method may be confused if that interval contains more than one root.

• `newton`, which finds the root near a specified starting point either by the secant method or by Newton’s method. The command is invoked with a statement like

```
q = so.newton(f, x0, fprime)
```

where `f` and the optional `fprime` are the names of functions that return the value and first derivative of the function whose root is sought and `x0` is the initial guess, presumably close to the desired root. (If `fprime` is not specified, `newton` invokes the secant method; otherwise, `newton` invokes Newton’s method.) The command also admits a number of keywords (with defaults), including

- `args=()`, which provides a tuple containing values to be passed as arguments to `f` and `fprime`.
- `tol=1.48e-08`, which specifies the allowable error in the returned root.
- `maxiter=50`, which provides an upper limit on the number of iterations before a message of non-convergence will be displayed.

### 14.10.1 Classical Turning Points

We might, for example, find the turning points for the potential energy in Eq. (14.3) with the PYTHON statements
14.10. FINDING ROOTS NUMERICALLY WITH PYTHON

\[
A = [\ 1.0/10000.0, \ 1.0/200.0, \ -1.0/500.0, \ -0.5 \ ]
\]
\[
q = \text{np.roots}( A )
\]
\[
\text{print( } q \text{ )}
\]
\[
[-48.26826757 \ -11.08043663 \ 9.3487042]
\]

All three roots have turned out to be real.

Other methods for finding the required roots all require that the function be defined in a py-file. For this example, we might construct the file

```python
def turnpts(x):
    # TURNPTS: returns values for a particular potential energy.
    fct = x**3/10000.0+x**2/200.0-x/500.0 - 0.5
    return fct
```

which we can store in the default directory with the name `turnpts.py`. Alternatively, we can provide these statements interactively at the PYTHON prompt in either the command window or the Python Shell.

To invoke the method of bisection, we also need some information about the location of each root in order to start the process. Basing our input on the information extracted from Fig. 14.1, we would find the roots of the defined polynomial with the PYTHON statements

```python
execfile('turnpts.py') or exec(open('turnpts.py').read() )
q1 = so.bisect( turnpts, -50.0, -40.0 )
q2 = so.bisect( turnpts, -12.0, -10.0 )
q3= so.bisect( turnpts, 9.0, 10.0 )
print([q1, q2, q3 ])
[-48.26826756675132, -11.080436630849363, 9.348704197598636]
```

We can invoke the secant method with only one starting value for each root, in which case the statements

```python
q1 = so.newton( turnpts, -50.0 )
q2 = so.newton( turnpts, -10.0 )
q3 = so.newton( turnpts, 10.0 )
print([q1, q2, q3 ])
[-48.26826756675163, -11.080436630848418, 9.348704197600052]
```

will yield the desired results.

Finally, to invoke Newton’s method, we must provide also the function `turnptsp` defined with the statements

```python
def turnptsp( x ):
    fctp = 3*x**2/10000.0+2*x/200.0-1/500.0
    return fctp
```

to return the derivatives of this function. The statements
execfile('turnptsp.py') or exec(open('turnptsp.py').read() )
q1 = so.newton( turnpts, -50.0, turnptsp )
q2 = so.newton( turnpts, -10.0, turnptsp )
q3 = so.newton( turnpts, 10.0, turnptsp )
print( [q1, q2, q3 ] )
[-48.268267566751625, -11.080436630848418, 9.348704197600052]

then yield the desired roots.

The results obtained by different methods are all the same to within about \( \pm 1 \times 10^{-8} \) but differ in the digits beyond that point, a conclusion entirely consistent with the default absolute errors described at the beginning of Section 14.10.

14.10.2 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 \texttt{roots}. We seek the roots as functions of \( \kappa \), which we shall represent by the variable \( \text{kvals} \) in our statements to PYTHON. 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 \( \pi \) over the interval \( 0.0 \leq \pi \leq 4.0 \), 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 \( \pi \). Let us therefore begin by creating two lists, the first of which contains the 41 values of \( \pi \) 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 \( \pi \). These ends are achieved with the statements

\[
\begin{align*}
kvals &= \text{np.linspace}(0.0,4.0,41) \\
\text{coeff1} &= -2.0*(1.0+kvals) \\
\text{coeff2} &= 1.0+2.0*kvals
\end{align*}
\]

which set the values of \( \pi \) and then create two 41-element lists, each containing the 41 value as of one of the coefficients. Finally, we find the roots and concatenate them into a single array with the statements

\[
\begin{align*}
s &= [] \\
\text{for } i \text{ in } \text{np.arange}(41): \\
& \quad s = \text{np.append}(s, [1,coeff1[i], coeff2[i] ] ) \\
s &= s.\text{reshape}(41,3) \\
s1 &= [] \\
\text{for } i \text{ in } \text{np.arange}(41): \\
& \quad s1 = \text{np.append}( s1,\text{np.roots}( s[i] ) )
\end{align*}
\]

The result is a 2 column \( \times \) 41 row array of real values, the first column containing the first root for each value of \( \pi \) and the second column containing the second root. 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 \texttt{roots} and the other contained in the second column—with the statements

\[
\begin{align*}
\text{rts} &= \text{np.sqrt}( s1.\text{reshape}(41,2) ) \\
\text{plt.plot} (kvals, rts, \text{linewidth}=3, \text{color}='\text{black}') \\
\text{plt.xlabel}( '$\kappa$', \text{fontsize}=14 ) \\
\text{plt.ylabel}( '$\omega$', \text{fontsize}=14 )
\end{align*}
\]
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.

```python
plt.grid()
plt.ylim([0.0, 3.0])
plt.tick_params(labelsize=12)
plt.show()
```

The resulting graph is shown in Fig. 14.9.

### 14.10.3 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 $\alpha = \frac{bv_0}{mg}$ 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). Central to the entire calculation is the py-file listed in Table 14.1, which we store in the default directory in a file named `proj.py`. For example, the upper left frame in Fig. 14.4 could have been created by setting several preliminary variables with the statements

```python
thetavals = np.pi*np.linspace(0.0, 6.0, 7)/12.0  # Set $\theta = 0^\circ, 15^\circ, \ldots, 90^\circ$ in radians.
tau = np.linspace(0.0, 2.0, 201)  # Set values of $\tau$.
```

and then executing the statements
def proj( tau, alpha, theta ):
    # PROJ: Evaluates function involved in projectile motion.
    # The function proj returns the value of a dimensionless
    # function that plays a role in determining the range of
    # a projectile fired at an elevation theta into a medium
    # whose viscosity is characterized by the dimensionless
    # parameter alpha. These parameters must be provided using
    # the keyword args.
    tmp = 1.0+alpha*np.sin(theta) # Calculate coefficient
    fct = tau - tmp*(1-np.exp(-tau)) # Return value of f(tau)
    return fct

eexecfile('proj.py') or
exec(open('proj.py').read() )
for i in np.arange(0,7):
    plt.plot( tau, proj( tau, 0.1, thetavals[i] ), 
              color='black', linewidth=3.0 );
plt.xlim( [0.0,0.25] )
plt.ylim( [-0.006,0.006] )
plt.tick_params(labelsize=12)
plt.grid()
plt.show()

Define function. See Section 5.7.
Plot graphs for $\alpha = 0.1$.
Set scaling, size of labels.
Turn on grid.
Display graph.

The remaining frames in Fig. 14.4 were produced with similar statements.

For this example, we choose PYTHON’s built-in routine newton, stipulating no function for
the derivative in calling the function so the secant method will be used for finding roots. Starting
with a fresh PYTHON 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

```
alpha = 0.4; theta = np.pi*30.0/180.0
tau0 = 0.2
q = so.newton(proj, tau0, args=(alpha,theta) )
print(q)
```

```
0.37643799724946136
```

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

```
range = np.cos(theta) * ( 1-np.exp(-q) ) / alpha
```

```
0.6791768095163329
```

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 increments of $1^\circ$ between

---

13 Over the initially specified interval in which the root is to be sought, the function must change sign. Consequently, for this function (see Fig. 14.4), we cannot specify a range that begins at 0.0, since the function does not have opposite signs at the two ends of that interval.
consecutive values of $\theta$. Continuing in the same session with PYTHON, we prepare for a calculational loop with the statements\(^{14}\)

\[
\text{thetadeg} = \text{np.linspace}(0.0, 90.0, 91 ) \quad \text{Set values of } \theta \text{ in degrees.}
\]

\[
\text{thetarad} = \text{np.pi*thetadeg/180.0} \quad \text{Convert } \theta \text{ to radians.}
\]

Then, choosing $\alpha = 0.4$ and noting from the first frame in Fig. 14.4 that all roots lie between $\tau = 0.1$ and $\tau = 1.0$, we continue with the statements

\[
\text{soln} = [ ] \quad \text{Prepare list for values. Set } \alpha.
\]

\[
\text{alpha} = 0.4 \quad \text{Set initial guess for root.}
\]

\[
\text{tau0} = 0.4 \quad \text{Find root for each } \theta.
\]

for i in np.arange(0,91):
    \[
    \text{theta} = \text{thetarad}[i] \quad \text{Set initial guess for root.}
    \]
    \[
    \text{rt} = \text{sc.newton( proj, tau0, args=(alpha, thetarad[i]) )} \quad \text{Find root for each } \theta.
    \]
    \[
    \text{soln} = \text{np.append(soln, rt) } \quad \text{Evaluate range for each root.}
    \]
    \[
    \text{range4} = \text{np.cos(thetarad)*(1-np.exp(-soln))/alpha} \quad \text{Evaluate range for each root.}
    \]

Repeating this process with ($\alpha = 0.1, \tau_0 = 0.4$), with ($\alpha = 0.2, \tau_0 = 0.3$), and with ($\alpha = 0.8, \tau_0 = 0.7$), storing the results in $\text{range1}$, $\text{range2}$, and $\text{range8}$, respectively, we obtain the data to plot Fig. 14.10 with the statements

\[
\text{plt.plot( thetadeg, range1, color='black', linewidth=3.0 )}
\]

\[
\text{plt.plot( thetadeg, range2, color='black', linewidth=3.0 )}
\]

\[
\text{plt.plot( thetadeg, range4, color='black', linewidth=3.0 )}
\]

\[
\text{plt.plot( thetadeg, range8, color='black', linewidth=3.0 )}
\]

\[
\text{plt.grid()}
\]

\[
\text{plt.xlabel('$\theta$ (deg)', fontsize=16 )}
\]

\[
\text{plt.ylabel('$R(\theta)$', fontsize=16 )}
\]

\[
\text{plt.xlim( [0.0,90.0] ); plt.ylim( [0.0,1.0] )}
\]

\[
\text{plt.tick_params(labelsize=12) plt.show()}
\]

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 $\text{range}n$ to find the maximum range and the approximate firing angle to achieve it in each case. For example, the statements

\[
\text{tmp} = [ \text{np.argmax( range1), np.argmax( range2), } \]
\[
\text{np.argmax( range4), np.argmax( range8) ]}
\]

\[
\text{print( tmp )}
\]

\[
[44, 43, 40, 37]
\]

reveal the index of the maximum value in each of the four ranges (which are also the angles of launch to achieve that range), and the statement

\(^{14}\)Because the roots for the lowest few values of $\theta$ are difficult to find, a bit of exploration was necessary before discovering that a successful evaluation would emerge only if the evaluations were started at $\theta = 10^\circ$.\
Figure 14.10: Range versus $\theta$ for $\alpha = 0.1$ (highest graph), $\alpha = 0.2$, $\alpha = 0.4$, and $\alpha = 0.8$ (lowest graph).

then displays the maximum value and the values on either side of that maximum in range1. Fitting a parabola to the three points and finding the coordinate of its maximum yields the angle 43.71°. Similar statements applied to the other three values of $\alpha$ yield that the maximum range (in units of $v_0^2/g$) and firing angle are

- 0.91384 and about 44° (43.71°) for $\alpha = 0.1$,
- 0.84133 and about 43° (42.53°) for $\alpha = 0.2$,
- 0.72602 and about 40° (40.43°) for $\alpha = 0.4$, and
- 0.56957 and about 37° (37.01°) for $\alpha = 0.8$.

14.10.4 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 `qmupper` containing the lines

```python
def qmupper( s ):
    # QMUPPER: Returns the value of sin(s)-0.04*s
    return np.sin(s) - 0.04*s
```

and the function `qmlower` containing the lines

```python
[ range1[43], range1[44], range1[45] ]
[0.9135634154637458, 0.9137922256264706, 0.9129322442946748]
```

See an exercise towards the end of the chapter(s) on specific symbol manipulating programs.
14.10. FINDING ROOTS NUMERICALLY WITH PYTHON

```python
def qmlower( s ):
    # QMLOWER: Returns the value of sin(s)+0.04*s
    return np.sin(s) + 0.04*s

and storing the two function definitions in the file qm.py in the default directory. Then, taking the limits as given in Eqs. (14.26) and (14.27), we find the roots in each category with the PYTHON statements

```python
upper=np.zeros(8)
upper[0] = so.bisect( qmupper, -0.5, 0.5 )
upper[1] = so.bisect( qmupper, 2.5, 3.5 )
    .
upper[7] = so.bisect( qmupper, 20.5, 21.5 )
lower= np.zeros(9)
lower[0] = so.bisect( qmlower, -0.5, 0.5 )
lower[1] = so.bisect( qmlower, 3.0, 4.0 )
    .
lower[8] = so.bisect( qmlower, 23.5, 24.5 )
```

and display them with the statements

```python
print( upper )
[ 0. 3.02047766 6.54820492 9.05418582 13.11879409 15.06138916
  19.7610887 20.99428643]
print( lower )
[ 0. 3.27288492 6.03920377 9.8288367 12.06284802 16.42478429
  18.04325707 23.1777839 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 PYTHON are

```python
print(upper*np.cos(upper)/np.sin(upper) + np.sqrt(25.0**2-upper**2))
[ nan -3.72452291e-10 4.82543682e+01 7.26387839e-11 4.25627650e+01
  5.19762011e-11 3.06267447e+01 -1.52233781e-12]
print(lower*np.cos(lower)/np.sin(lower) + np.sqrt(25.0**2-lower**2))
[ nan 4.95696772e+01 7.44648787e-11 4.59736433e+01 -1.76214598e-12
  3.76949047e+01 -5.17594856e-11 1.87393264e+01 -2.59428035e-11]
```

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 PYTHON statements

```python
```
Discard first root in lower.
Replace “bad” roots in \( s \) with “good” ones in \( u \).
Display “good” roots.

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

\[
\text{energy} = -(1 - s**2./25.0**2)
\]

The statements

\[
\text{for } i \text{ in np.arange(8):}
\]

\[
\text{plt.plot( } [0.2, 1.0], [\text{energy}[i], \text{energy}[i]], \text{ color='black',}
\]

\[
\text{linestyle=2.0 )}
\]

\[
\text{plt.box(on=None)}
\]

\[
\text{plt.plot( } [0.0, 1.2, 1.2, 1.5], [-1.0, -1.0, 0.0, 0.0], \text{ color='black'}
\]

\[
\text{plt.plot( } [0.0, 0.0], [-1.5, 0.5], \text{ color='black'}
\]

\[
\text{plt.xlabel('$E/V_0$', fontsize=16)}
\]

\[
\text{plt.tick_params(axis='x', which='both', bottom=False, top=False,}
\]

\[
\text{labelbottom=False )}
\]

\[
\text{plt.tick_params(labelsize=14)}
\]

will produce the energy level diagram of Fig. 14.11. The for-loop opens the plot window and draws the energy levels, plt.box turns off the enclosing box, and the remaining statements draw the potential well and arrange for there to be only a labeled vertical axis.

14.13 Finding Roots Numerically with Mathematica

When *Mathematica* is able to find an analytic solution for the roots of an equation of interest, numerical values of the resulting solution can be found simply by embedding the command Solve in the command N in a statement of the form

\[
\text{N[ Solve[ Eqn, Var ] ]}
\]

Here, *Eqn* stands for the equation to be solved and *Var* represents the variable to be found. If, however, *Mathematica* is unable to find an analytic solution (or we want to go directly to a numerical approach anyway), we must resort to numerical methods. *Mathematica*'s workhorse when the function whose roots are sought is a polynomial is the command NSolve. The syntax for NSolve is essentially identical to that of the command Solve. In its simplest form, we merely invoke the function in a statement like

\[
\text{NSolve[ Eqn, Var ]}
\]

though we might also exploit a further argument in a statement of the form

\[
\text{NSolve[ Eqn, Var, NumberOfDigits ]}
\]

to stipulate a precision different from the default. More specifically, the statement
14.13. FINDING ROOTS NUMERICALLY WITH MATHEMATICA 445

Figure 14.11: Energy level diagram for the one-dimensional quantum well when $c = 25.0$. The heavy lines show the energies, measured in units of $V_0$; the light lines show the energies of the bottom ($-1$) and the top ($0$) of the well.

\[
\begin{align*}
\text{In[1]} &:= \text{NSolve}[ 3x^3-2x+x-4==0, x ] \\
\text{Out[1]} &= \{ 
{x \to -0.600668 - 0.865491i}, 
{x \to -0.600668 + 0.865491i}, 
{x \to 1.20134}\}
\end{align*}
\]

returns numerical values for the three roots of the specified cubic polynomial, including both real and complex roots. Despite its nominal restriction to polynomials, however, \texttt{NSolve} is able to find roots for some non-polynomial functions. For example, the statement

\[
\begin{align*}
\text{In[2]} &:= \text{NSolve}[ \cos(x)==0, x ] \\
\text{Out[2]} &= \{ 
{x \to \text{ConditionalExpression}[1.\left(-1.5708 + 6.28319C[1]\right), C[1] \in \text{Integers}]} , 
{x \to \text{ConditionalExpression}[1.\left(1.5708 + 6.28319C[1]\right), C[1] \in \text{Integers}]} \}
\end{align*}
\]

reveals that \textit{Mathematica} recognizes the equation $\cos(x) = 0$ has many roots and successfully if obscurely reports them all. To obtain any particular roots, we would execute statements like

\[
\begin{align*}
\text{In[3]} &:= \text{Out[2]} /. \ C[1] \rightarrow 0 \\
\text{Out[3]} &= \{ 
{x \to -1.5708}, 
{x \to 1.5708}\}
\end{align*}
\]

\[
\begin{align*}
\text{In[4]} &:= \text{Out[2]} /. \ C[1] \rightarrow 3 \\
\text{Out[4]} &= \{ 
{x \to 17.2788}, 
{x \to 20.4204}\}
\end{align*}
\]

$C[1]$ can, of course, be given either positive or negative integer values. Apparently, when \texttt{NSolve} is presented with a non-polynomial, it reverts to \texttt{Solve} followed by \texttt{N}, and the effort will be successful only if the given equation admits an analytic solution.\footnote{Some earlier versions of \textit{Mathematica} found only two of the many roots of $\cos(x) = 0$, the command making its own judgment about the interval in which to search and offering no means for us to specify some other interval. For the particular equation $\cos(x) = 0$, those versions of \textit{Mathematica} also displayed the warning “Solve::ifun: Inverse functions are being used by Solve, so some solutions may not be found.”}
When the function is not a polynomial, when it does not possess an analytic solution, or when we seek a numerically determined solution anyway, we must use the command \texttt{FindRoot}, which has arguments specifying not only the equation to be solved and the variable sought but also one or two starting values for an iterative search. The general syntax is

\[
\text{FindRoot} \left[ \text{Eqn}, \{ \text{Var}, \text{Start1} \} \right]
\]

in which case \texttt{FindRoot} uses Newton’s method, or

\[
\text{FindRoot} \left[ \text{Eqn}, \{ \text{Var}, \text{Start1}, \text{Start2} \} \right]
\]

in which case \texttt{FindRoot} uses the secant method.\footnote{Remember that the two starting values given to the secant method are not necessarily lower and upper bounds for the root that will be found. If they do happen to bound a root, however, that is the root that will be found.} \texttt{FindRoot} admits several options, the most significant of which are \texttt{MaxIterations} (default 100) to specify the maximum number of iterations to use before giving up and \texttt{AccuracyGoal} (default normally half of the value of \texttt{WorkingPrecision})\footnote{The default for \texttt{WorkingPrecision} is \texttt{$\$MachinePrecision}, often 16.} to stipulate the accuracy with which the function will be zero at the returned root. Thus, for example, we might seek a particular root of \(\cos(x)\) with the statement

\[
\text{In}[3]:= \text{FindRoot} \left[ \cos(x) == 0, \{x, 4.0\} \right]
\]

\[
\text{Out}[3]= \{\{x \rightarrow 4.71239\}\}
\]

or the statement

\[
\text{In}[4]:= \text{FindRoot} \left[ \cos(x) == 0, \{x, 4.0, 5.0\} \right]
\]

\[
\text{Out}[4]= \{\{x \rightarrow 4.71239\}\}
\]

a value we might recognize as \(3\pi/2\). Properly chosen, the additional stipulations in the second argument permit us to select a particular root from the many that we know \textit{a priori} exist.

Out of curiosity, let us test \texttt{NSolve} with a polynomial that has a multiple root. We execute the statements

\[
\text{In}[5]:= p1 = \text{Expand} \left[ (x-1)^2 \cdot (x+3) \cdot (x-4) \right]; \quad \text{Create polynomial with multiple roots.}
\]

\[
\text{In}[6]:= \text{NSolve} \left[ p1==0, x \right]; \quad \text{Find roots with \texttt{NSolve}.}
\]

\[
\text{Out}[6]= \{\{x \rightarrow -3.\}, \{x \rightarrow 1.\}, \{x \rightarrow 4.\}\}
\]

Clearly, \texttt{NSolve} has returned all four roots of the given polynomial even though one of the three distinct roots is a double root.

14.13.1 Classical Turning Points

To find the turning points for the potential energy in Eq. (14.3), we simply define the polynomial and invoke \texttt{NSolve} with the statements

\[
\text{In}[7]:= V = x^3/10000 + x^2/200 - x/500 - 1/2;
\]

\[
\text{In}[8]:= \text{NSolve} \left[ V == 0, x \right]
\]

\[
\text{Out}[8]= \{\{x \rightarrow -48.2683\}, \{x \rightarrow -11.0804\}, \{x \rightarrow 9.3487\}\}
\]

\[
\text{In}[9]:= \text{Quit}[]
\]

These results are certainly consistent with the bounds identified in Section 14.1.1.
14.13.2 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 \( \alpha = b v_0/(mg) \) 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). To find that root and then the range for a particular choice of \( \alpha \) and \( \theta \), we begin with the statements

\[
\text{In}[1]:= \text{rang} = \frac{(1 - e^{-\tau}) \cos[\theta]}{\alpha}
\]

\[
\text{Out}[1] = \frac{(1 - e^{-\tau}) \cos[\theta]}{\alpha}
\]

\[
\text{In}[2]:= \text{fct} = \tau - (1 - \alpha \sin[\theta]) \cdot (1 - e^{-\tau})
\]

\[
\text{Out}[2] = \tau - (1 - \alpha \sin[\theta]) \cdot (1 - e^{-\tau})
\]

which define the range given by Eq. (14.19) and the auxiliary function given by Eq. (14.18). Then, we proceed with the statements

\[
\text{In}[3]:= \alpha = 0.4; \theta = \text{Pi} \cdot 30.0/180.0; \quad \text{Choose } \alpha, \theta.
\]

\[
\text{In}[4]:= \text{FindRoot}[\text{fct} == 0, \{\tau, 0.2, 0.5\}] \quad \text{Find root.}
\]

\[
\text{Out}[4] = \{\tau \to 0.376438\}
\]

\[
\text{In}[5]:= \text{rang} /. \%
\]

\[
\text{Out}[5] = 0.679177
\]

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° to 90°, choosing, say, increments of 1° between consecutive values of \( \theta \). To find the range for each of the selected values of \( \theta \), we invoke the single loop

\[
\text{In}[6]:= \text{For}[i = 0, i < 91, i++,
\quad \theta = \text{Pi} \cdot i/180.0;
\quad \text{tmp} = \text{FindRoot}[\text{fct} == 0, \{\tau, 0.005, 0.8\}];
\quad \text{rng}[i+1] = \text{rang} /. \text{tmp}
\]

where we have accepted the default value 100 of \texttt{MaxIterations}. (Remember that \( \alpha \) was set to 0.4 in the previous paragraph.) The starting values are set with reference to Fig. 14.4 to prevent \texttt{FindRoot} from finding the root \( \tau = 0 \) (which exists for all values of \( \theta \)). Unfortunately, that strategy doesn’t work for \( \theta = 0 \), at which value the only root is \( \tau = 0 \). Even though no warning is presented, \texttt{FindRoot} returns an incorrect value for the root at \( \theta = 0 \), which we must correct with the statement

\[
\text{In}[7]:= \text{rng}[1] = 0.0;
\]

after the above loop has terminated. Finally, we produce the graph in Fig. 14.12 with the statements

\[
\text{In}[8]:= \text{pts} = \text{Table}[\{i, \text{rng}[i+1]\}, \{i, 0, 90\}];
\]

\[
\text{In}[9]:= \text{ListPlot}[\text{pts}, \text{PlotJoined} \to \text{True},
\quad \text{AxesLabel} \to \{"theta", "R(theta)"},
\quad \text{PlotStyle} \to \text{Thickness}[0.005]\]
\]

Further, by noting the three values,
we conclude that the maximum range for $\alpha = 0.4$ occurs near $^{19} \theta = 40^\circ$ (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.

14.13.3 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 create lists of the lower and upper bounds given in Eqs. (14.26) and (14.27). To that end we execute the statements$^{20}$

\begin{verbatim}
In[1]:= ulb = { -0.5, 2.5, 6.0, 8.5, 12.5, 14.5, 19.0, 20.5 }; 
In[2]:= uub = { 0.5, 3.5, 7.0, 9.5, 13.5, 15.5, 20.0, 21.5 }; 
In[3]:= llb = { -0.5, 3.0, 5.5, 9.5, 11.5, 16.0, 17.5, 22.5, 23.5, 22.5, 24.5 }; 
In[4]:= 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

$^{19}$Remember that the index of the entries in rng is one larger than the angle in degrees to which that entry corresponds.

$^{20}$We use a three-character variable name, where the first character will be u or l for the upper or lower sign in Eq. (14.25) and the remaining characters will be lb or ub for the lower or upper bounds on the root. Thus, for example, ulb will contain the lower bounds for the roots associated with the upper sign.
In[5]:= For[ i = 1, i<9, i++,
    rootu[i] = FindRoot[ Sin[s]-0.04*s, { s, ulb[[i]], uub[[i]]} ] ]
In[6]:= For[ i = 1, i<10, i++,
    rootl[i] = FindRoot[ Sin[s]+0.04*s, { s, llb[[i]], lub[[i]]} ] ]

Next, remembering that the values in rootu[...] and rootl[...] have the form \( s \rightarrow \ldots \), we concatenate the elements into lists and display them with the statements

In[7]:= rtu = s /. Table[ rootu[i], { i, 1, 8 } ]
Out[7]=
In[8]:= rtl = s /. Table[ rootl[i], { i, 1, 9 } ]
Out[8]=
\{ 0., 3.27288, 6.0392, 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 Mathematica are

In[9]:= f[x_] := x*Cot[x] + Sqrt[25^2-x^2]
In[10]:= Map[ f, rtu ]
Out[10]=
\{ \text{Indeterminate}, -1.42109 \times 10^{-14}, -1.482544, -3.55271 \times 10^{-15}, \\
42.5628, 3.55271 \times 10^{-14}, 30.6267, -4.79616 \times 10^{-14} \}
In[11]:= Map[ f, rtl ]
Out[11]=
\{ \text{Indeterminate}, 49.5697, 3.55271 \times 10^{-15}, 45.9736, -2.84217 \times 10^{-14}, \\
37.6949, 8.31335 \times 10^{-13}, 18.7393, 2.64038 \times 10^{-11} \}

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 Mathematica statements

In[12]:= s = Rest[ rtl ];
In[13]:= For[ i=1, i<8, i=i+2,
    s[[i]] = rtu[[i+1]] ]
In[14]:= s
Out[14]=

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

In[15]:= energy = -(1-s^2/25.0^2)
Out[15]=
\{ -.985403, -.941645, -.868835, -.76718, -.637047, -.479105, -.294784, -.088774 \}

Then, the statement

\footnote{Remember that values close to zero often should be exactly zero but fail in that regard because of internal roundoff. These values may differ from operating system to operating system.}
CHAPTER 14. FINDING ROOTS

In[16]:= pts = Table[ { {0.2,energy[[i]]}, {0.8,energy[[i]]} }, {i,8} ]
Out[16]= 
\{
\{ {0.2, -0.985403}, {0.8, -0.985403} \}, ...
\}

creates a list, each of whose elements is itself a list of the two points at opposite ends of a single line in an energy level diagram for the system we are studying. Finally, we produce the energy level diagram itself in steps. First, we draw the axes and the lowest energy level with the statement

In[17]:= lv[1] = ListPlot[ pts[[1]], PlotJoined -> True, 
PlotRange -> {{0.0,1.0},{-1.5,0.5}}, 
PlotStyle -> Thickness[0.005], AxesOrigin -> {0.0,-1.5}, 
GridLines -> {{},{-1.0,-0.5,0.0,0.5}, 
AxesLabel -> {"", "E/V0"} ]

Then, we create several graphic objects, each containing one of the energy levels, with the statement

In[18]:= For[ i=2, i<9, i++, 
   lv[i] = Graphics[ { Thickness[0.005], Line[ pts[[i]]] } ] ]

Finally, we concatenate the graphics objects into a single list and create the display with the statement

In[19]:= Show[ Table[ lv[i], {i, 8} ] ]

The resulting diagram is shown in 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 $HEAD/fortran, where (as defined in the Local Guide) $HEAD 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 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 14.2. After storing this

22 The list presented as an argument to the command Graphics may contain graphics primitives, e.g., Line, and graphics directives, e.g., Thickness. The order is important, since the directive affects only those primitives that follow it in the list.
14.14. FINDING ROOTS NUMERICALLY WITH FORTRAN

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 the energies of the bottom \((-1)\) and the top \((0)\) of the well.

![Energy level diagram](image)

Table 14.2: The FORTRAN program `GUESSROOT.F`.

```fortran
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
```

program in the file `guessroot.f`, we compile, link, and run it with the two statements\(^{23}\)

```
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,\(^{23}\)

\(^{23}\)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*.\]
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."24

```
./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 = -48.2735 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, 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.A. This program (1) defines the function whose roots will be sought with the lines

```
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

```
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

```
24The prompts, entered values, and returned values have been displayed on the same line to save space.
```
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 \textit{exactly} zero. Using this program, which we compile, link, and run with the statements

```
f77 -o bisect.xf bisect.f
./bisect.xf
```

we have the conversation

```
./bisect.xf
Lower bound = -50.0
Upper bound = -40.0
ROOT = -48.2682E+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 \texttt{rtbis} (bisection), \texttt{rtnewt} (Newton's method), and \texttt{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 \texttt{rtbis.f}, which is invoked with a statement of the form

\begin{verbatim}
ROOT = RTBIS( FUNC, SLB, SUB, EPS )
\end{verbatim}

where \texttt{FUNC} defines the univariate function whose roots are to be found, \texttt{SLB} and \texttt{SUB} specify the bounds within which the desired root is known to lie, and \texttt{EPS} specifies the desired absolute accuracy.

A FORTRAN driving program using \texttt{RTBIS} must be written by the user and will have the general form shown in Table 14.3. This program defines the function whose roots are desired, establishes the bounds and the desired error for a given root, invokes \texttt{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 \texttt{xrtbis.f} to the default directory from the directory \texttt{$NRHEAD/recipes_f/demo/src}, giving it the name \texttt{sinrtbis.f}, and editing it appropriately or (2) by creating the file \texttt{sinrtbis.f} with an available text editor.

2. Copy the recipe \texttt{rtbis.f} from the directory \texttt{$NRHEAD/recipes_f/recipes} to the default directory.

3. Compile and link the executable module with the statement

\begin{verbatim}
f77 -o sinrtbis.xf sinrtbis.f rtbis.f
\end{verbatim}
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.25

```
./sinrtbis.xf
Lower bound on root = -0.5
Upper bound on root = 0.5
Error limit = 0.0001
Root is 0.00000
```

```
./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
```

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

```
ROOT = RTNEWT( FUNC, SLB, SUB, EPS)
```

where `FUNC` defines a subroutine that returns both the function whose roots are to be found and its
first derivative, `SLB` and `SUB` specify the bounds within which the desired root is known to lie, and
`EPS` specifies the desired absolute accuracy. A suitable driving program to use `rtnewt` to find the
roots of our test function differs from `sinrtbis` in only a few respects:

- The first line would name the program `sinrtnewt`.
- The line invoking `RTBIS` should invoke `RTNEWT` instead.
- The user-defined `function` must be replaced with a subroutine that returns both the value of
  the function and the value of its derivative. Thus, the function `FUNC` in `sinrtbis` must be
  replaced with the subroutine

---

25To save space, prompting messages and values input are shown on the same line.
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 xrtnewt.f from the directory $NRHEAD/recipes.f/demo/src, giving it the name sinrtnewt.f and editing it appropriately or (2) by creating the file sinrtnewt.f with an available text editor.

2. Copy the recipe rtnewt.f from the directory $NRHEAD/fortran/recipes.f/recipes to the default directory.

3. Compile and link the executable module with the statement

   f77 -o sinrtnewt.xf sinrtnewt.f rtnewt.f

4. Run the program with the statement ./sinrtnewt.xf.

The lowest two and the highest non-negative roots of the sample function would then be found with the statements²⁶

./sinrtnewt.xf
Lower bound on root = -0.5
Upper bound on root =  0.5
Error limit      =  0.0001
Root is          0.000000

./sinrtnewt.xf
Lower bound on root =  2.5
Upper bound on root =  3.5
Error limit      =  0.0001
Root is          3.020478

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

²⁶To save space, the prompting messages and the responses are shown on the same line.
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.

14.15.1 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 14.4. 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 proposals, until the desired accuracy is achieved.

---

27 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.

28 The 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.

29 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 14.4: The C program guessroot.c.

```c
/* PROGRAM guessroot.c */
#include <stdio.h> /* Load standard i.o routines */
#include <math.h> /* Load standard math routines */

main()
{
    float x, f; /* For trial root, function */
    printf( "First guess : " ); scanf( "%f", &x );
    while( x != 9999 ){
        f = pow(x,3)/10000.0 + pow(x,2)/200.0 - x/500.0 - 0.5;
        printf( "f(x) = %15.6f
", f );
        printf( "Next guess : " ); scanf( "%f", &x ) ;
    }
}
```

guesses, each guided by the results of all previous trials. We might, for example, find this root with the "conversation".

```
./guessroot.xc
First guess : -50.0 f(x) = -0.400000
Next guess : -49.0 f(x) = -0.161900
Next guess : -48.0 f(x) = 0.056800
Next guess : -48.5 f(x) = -0.050163
Next guess : -48.25 f(x) = 0.003911
Next guess : -48.375 f(x) = -0.022977
Next guess : -48.2813 f(x) = -0.002794
Next guess : -48.2656 f(x) = 0.000572
Next guess : -48.2735 f(x) = -0.001121
Next guess : -48.2695 f(x) = -0.000264
Next guess : -48.2676 f(x) = 0.000143
Next guess : -48.2685 f(x) = -0.000050
Next guess : 9999
```

Starting at a point we suspect to be below the root, we scan upwards until the function changes sign. In that way, we discover that the root lies between $x = -49.0$ and $x = -48.0$. From this point, we carry out a binary search, repeatedly halving the interval and deciding on the basis of the signs at each step which half of the current interval to refine further. By the time we stopped, we had concluded that the root lies somewhere between $x = -48.2676$ and $x = -48.2685$, i.e., that the root is $x = -48.268$ to three places after the decimal.

An algorithm to implement the method of bisection in a C program merely automates the decision making process that was explicit in the previous paragraph. A full listing of a possible program, which we have named bisect.c, is presented in Section 14.B. Beyond incorporating the necessary header files, this program (1) defines the function whose roots will be sought with the lines

---

30The prompt, the entered values, and the returned values have been displayed on the same line to save space.
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

tol = 0.0001
printf( "\nLower bound = " ); scanf( "%f", &xa );
printf( "\nUpper bound = " ); scanf( "%f", &xb );

and finally (4) evaluates the function at the two bounds, verifies that a root exists in the specified interval (and terminates execution if not), refines the estimate of the root, and displays the final value of the root, its final bounds, and the function at the declared root with the lines

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

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

we have the conversation

./bisect.xc
Lower bound = -50.0
Upper bound = -40.0
root = -48.268204
xb - xa = 0.000076
function = 0.000014

and find that the lowest root of our sample polynomial is $x = -48.2682 \pm 0.0001$. If we instead use the lower and upper bounds $-15.0$ and $-5.0$, this program converges to $x = -11.0804 \pm 0.0001$. Finally, with lower and upper bounds $5.0$ and $15.0$, the program converges to $x = 9.3487 \pm 0.0001$. 

/* 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 );
}

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

\[ \text{root} = \text{rtbis}( \text{func}, \text{slb}, \text{sub}, \text{eps} ) \]

where \text{func} defines the univariate function whose roots are to be found, \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 C driving program using rtbis must be written by the user and will have the general form shown in Table 14.5. 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 \texttt{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 \texttt{xrtbis.c} to the default directory from the directory $\texttt{NRHEAD/recipes\_c-ansi/demo/src}$, giving it the name \texttt{sinrtbis.c}, and editing it appropriately or (2) by creating the file \texttt{sinrtbis.c} with an available text editor.

2. Copy the file \texttt{rtbis.c} from the directory $\texttt{NRHEAD/recipes\_c-ansi/recipes}$ to the default directory.

3. Copy the header files \texttt{nr.h} and \texttt{nrutil.h} to the default directory from the directory $\texttt{NRHEAD/recipes\_c-ansi/include}$.

4. Copy the utility subroutine \texttt{nrutil.c} to the default directory from the directory $\texttt{NRHEAD/recipes\_c-ansi/recipes}$.

5. Compile and link the executable module with the statement

   \[
   \texttt{cc -o sinrtbis.xc sinrtbis.c rtbis.c nrutil.c -lm}
   \]

6. Run the program with the statement \texttt{./sinrtbis.xc}.

The lowest two and the highest non-negative roots of the sample function would then be found with the statements\(^{32}\)

\[
\begin{align*}
\texttt{./sinrtbis.xc} \\
&\text{Lower bound on root} = -0.5 \\
&\text{Upper bound on root} = 0.5 \\
&\text{Error limit} = 0.0001 \\
&\text{Root is} \quad 0.000000
\end{align*}
\]

\[
\begin{align*}
\texttt{./sinrtbis.xf} \\
&\text{Lower bound on root} = 2.5 \\
&\text{Upper bound on root} = 3.5 \\
&\text{Error limit} = 0.0001 \\
&\text{Root is} \quad 3.020508
\end{align*}
\]

\[
\begin{align*}
\texttt{./sinrtbis.xf} \\
&\text{Lower bound on root} = 20.5 \\
&\text{Upper bound on root} = 21.5 \\
&\text{Error limit} = 0.0001 \\
&\text{Root is} \quad 20.994324
\end{align*}
\]

An alternative algorithm uses \textit{Newton's method} and is implemented in \texttt{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, \texttt{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 \texttt{rtnewt} is invoked with a statement of the form

\[^{31}\text{Alternatively, at some sites, the language directory may be \texttt{recipes\_c-kr} rather than \texttt{recipes\_c-ansi}.}\]

\[^{32}\text{To save space, prompting messages and values input are shown on the same line.}\]
CHAPTER 14. FINDING ROOTS

root=rtnewt( func, slb, sub, eps );

where func defines a procedure that returns both the function whose roots are to be found and its first derivative, slb and sub specify the bounds within which the desired root is known to lie, and eps specifies the desired absolute accuracy. A suitable driving program to use rtnewt to find the roots of our test function differs from sinrtbis in only a few respects:

- The first line would name the program sinrtnewt.
- The line invoking rtbis should invoke rtnewt instead.
- The user-defined function must be replaced with a procedure that returns both the value of the function and the value of its derivative. Thus, the function func in sinrtbis must be replaced with the procedure

```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

```
./sinrtnewt.xc
Lower bound on root = -0.5
Upper bound on root = 0.5
Error limit = 0.0001
Root is 0.000000
```

---

33 To save space, the prompting messages and the responses are shown on the same line.
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

\[
\text{zbrak}( \text{func}, a, b, n, \text{slb}, \text{sub}, \&\text{nroots})
\]

When called, zbrak divides the interval \(a \leq x \leq b\) into \(n\) equal segments, scans through those segments, identifies any in which the function func changes sign, and stores the lower and upper bounds of each such segment in slb[] and sub[]. On input nroots specifies the maximum number of such segments that can be stored; on return, nroots stores the actual number of such segments found in the specified interval. A suitable driving program must dimension slb and sub appropriately, invoke zbrak, and then invoke rtbis in a loop whose index steps from one bracket to the next. The demonstration programs $NRHEAD/recipes/c-ansi/demo/src/xrtbis.c, .../xrtnewt.c, and .../xrtsafe.c all illustrate this strategy. Note that zbrak will—erroneously—conclude that a root exists at a point on one side of which the function diverges to \(+\infty\) and on the other side of which the function diverges to \(-\infty\), so we must be cautious in using zbrak on functions that have singularities in the interval in which roots are sought.

## 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.\(^{34}\) 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.\(^{35}\)

---

\(^{34}\) 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.

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.,

$$
\begin{pmatrix}
  a_{11} & a_{12} & a_{13} \\
  a_{21} & a_{22} & a_{23} \\
  a_{31} & a_{32} & a_{33}
\end{pmatrix}
\begin{pmatrix}
  x_1 \\
  x_2 \\
  x_3
\end{pmatrix}
= 
\begin{pmatrix}
  b_1 \\
  b_2 \\
  b_3
\end{pmatrix}
$$

(14.37)

where, with $i$ and $j$ independently assuming the values 1, 2, and 3, the quantities $x_i$ are the unknowns, the quantities $b_i$ are the inhomogenieties, and the quantities $a_{ij}$ are the coefficients defining the equations. Here, we have three equations and three unknowns. More generally, for a system of $n$ equations and $n$ unknowns, we would have an $n \times n$ matrix $A$ of coefficients, an $n$-element vector $\mathbf{x}$ of unknowns and an $n$-element vector $\mathbf{b}$ of inhomogenieties, we would write the equations compactly in the form

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

(14.38)

and we would write their solution formally as

$$
\mathbf{x} = A^{-1} \mathbf{b}
$$

(14.39)

Less compactly but more usefully (at least occasionally), we might remember Cramer’s rule and write the solution in terms of determinants in the form

$$
x_1 = \frac{ \begin{vmatrix}
  b_1 & a_{12} & a_{13} \\
  a_{21} & a_{22} & a_{23} \\
  a_{31} & a_{32} & a_{33}
\end{vmatrix}}{ \begin{vmatrix}
  a_{11} & a_{12} & a_{13} \\
  a_{21} & a_{22} & a_{23} \\
  a_{31} & a_{32} & a_{33}
\end{vmatrix}} ; 
 x_2 = \frac{ \begin{vmatrix}
  a_{11} & b_1 & a_{13} \\
  a_{21} & a_{22} & a_{23} \\
  a_{31} & a_{32} & a_{33}
\end{vmatrix}}{ \begin{vmatrix}
  a_{11} & a_{12} & a_{13} \\
  a_{21} & a_{22} & a_{23} \\
  a_{31} & a_{32} & a_{33}
\end{vmatrix}} ; 
 x_3 = \frac{ \begin{vmatrix}
  a_{11} & a_{12} & b_1 \\
  a_{21} & a_{22} & a_{23} \\
  a_{31} & a_{32} & a_{33}
\end{vmatrix}}{ \begin{vmatrix}
  a_{11} & a_{12} & a_{13} \\
  a_{21} & a_{22} & a_{23} \\
  a_{31} & a_{32} & a_{33}
\end{vmatrix}}
$$

(14.40)

In this rule, which can be readily extended to $n$ equations, the denominator in the expression for the $i$-th unknown is the determinant of the coefficient matrix and the numerator is the determinant of the matrix created from the coefficient matrix by replacing its $i$-th column with the column of inhomogenieties. Cramer’s rule provides a direct, symbolic, and exact solution to the system of simultaneous linear equations. Note, however, that the rule gives problems if the determinant of the coefficient matrix happens to be zero or, equivalently, if the inverse $A^{-1}$ of the coefficient matrix fails to exist. When $|A| = 0$, the equations are said to be singular and will either have no solution (equations inhomogeneous; $\mathbf{b} \neq 0$) or an infinite number of solutions (equations homogeneous; $\mathbf{b} = 0$).

While compact, Cramer’s rule is not particularly useful for numerical solution of even modest sized systems, since the most direct approach to evaluating determinants is vulnerable to round off error.\(^{36}\) We can, however, invent alternative methods that are computationally more satisfactory. The simplest algorithm to describe involves Gaussian elimination, in which one variable at a time is systematically eliminated to yield a simpler system whose solution is readily found by a process called backsubstitution. We illustrate with Eq. (14.37), but the schema is readily extended to $n$ equations. The process of Gaussian elimination entails

\(^{36}\)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.
• dividing each equation by the coefficient of \( x_1 \), obtaining\(^{37} \)
\[
\begin{bmatrix}
1 & a'_{12} & a'_{13} \\
1 & a'_{22} & a'_{23} \\
1 & a'_{32} & a'_{33}
\end{bmatrix}
\begin{bmatrix}
x_1 \\
x_2 \\
x_3
\end{bmatrix}
=
\begin{bmatrix}
b'_1 \\
b'_2 \\
b'_3
\end{bmatrix}
\] (14.41)

• keeping the first equation and replacing the second and third with the result of subtracting the first from each in turn, obtaining
\[
\begin{bmatrix}
1 & a'_{12} & a'_{13} \\
0 & a'_{22} & a'_{23} \\
0 & a'_{32} & a'_{33}
\end{bmatrix}
\begin{bmatrix}
x_1 \\
x_2 \\
x_3
\end{bmatrix}
=
\begin{bmatrix}
b'_1 \\
b''_2 \\
b''_3
\end{bmatrix}
\] (14.42)

• dividing the second and third equations by the coefficient of \( x_2 \), obtaining
\[
\begin{bmatrix}
1 & a'_{12} & a'_{13} \\
0 & 1 & a''_{23} \\
0 & 1 & a''_{33}
\end{bmatrix}
\begin{bmatrix}
x_1 \\
x_2 \\
x_3
\end{bmatrix}
=
\begin{bmatrix}
b'_1 \\
b''_2 \\
b''_3
\end{bmatrix}
\] (14.43)

• keeping the first and second equations but replacing the third with the result of subtracting the second from the third, obtaining
\[
\begin{bmatrix}
1 & a'_{12} & a'_{13} \\
0 & 1 & a''_{23} \\
0 & 0 & a''_{33}
\end{bmatrix}
\begin{bmatrix}
x_1 \\
x_2 \\
x_3
\end{bmatrix}
=
\begin{bmatrix}
b'_1 \\
b''_2 \\
b''_3
\end{bmatrix}
\] (14.44)

In essence, Gaussian elimination converts the original system of equations with a general coefficient matrix into an equivalent system whose coefficient matrix is upper triangular. In that form, however, the solution is readily obtained by backsubstitution. The third equation tells us directly that \( x_3 = b''_3/a''_{33} \). Then, knowing \( x_3 \), we find from the second equation that \( x_2 = b''_2 - a''_{23}x_3 \) and, knowing \( x_3 \) and \( x_2 \), we find from the first equation that \( x_1 = b'_1 - a'_{12}x_2 - a'_{13}x_3 \). The job is done!

Unfortunately, in a computer whose arithmetic is done to finite precision, the order in which the equations are treated in this process and the order in which the variables are placed can have a significant impact on the quality of the solution obtained. Thus, while Gaussian elimination 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) conditions, 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.\(^{38} \) That is, we can write
\[
A = LU \quad \text{where} \quad L = \begin{bmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{bmatrix} \quad \text{and} \quad U = \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}
\] (14.45)

\(^{37} \)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.

This so-called \( LU \) decomposition allows us to seek the solution of the original equation in steps. First, we view the equation in the form

\[
Ax = b \quad \Rightarrow \quad LUx = b \quad \Rightarrow \quad Ly = b \quad \text{where} \quad y = Ux \quad (14.46)
\]

Since the first equation in the (matrix) equation \( Ly = b \) tells us \( y_1 \) directly once \( y_1 \) is known, and the second tells us \( y_2 \) 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 \textit{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 \textit{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 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—\textit{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.\(^{39}\)

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.

**MATHEMATICA:** Within \textit{Mathematica}, systems of linear equations can be solved analytically either with the special command \texttt{LinearSolve}, which accepts the matrix of coefficients and the vector of inhomogeneities as arguments, or with the command \texttt{Solve}, whose arguments then are lists of the equations and the unknowns. Numerical solutions can be found by nesting either \texttt{LinearSolve} or \texttt{Solve} in the command \texttt{N}. While \texttt{NSolve} cannot handle systems of equations, \texttt{FindRoot} can address systems of equations if we simply make its first argument a list of the equations to be solved, supply as its second argument a list containing the first variable and a starting guess for its value in the

solution, supply as its third argument a list containing the second variable and a starting guess for its value in the solution, etc.

**NUMERICAL RECIPES in FORTRAN:** For numerical solution of systems of linear equations, the FORTRAN subroutines in the *Numerical Recipes* library include gaussj.f, which uses Gauss-Jordan elimination (similar to Gaussian elimination); ludcmp.f and lubksb.f, which effect the LU decomposition of the coefficient matrix and the solution of the associated system; tridag.f, which is especially adapted to tridiagonal systems; and svdcmp.f and svbsb.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.

**NUMERICAL RECIPES in C:** For numerical solution of systems of linear equations, the C subroutines in the *Numerical Recipes* library include gaussj.c, which uses Gauss-Jordan elimination (similar to Gaussian elimination); ludcmp.c and lubksb.c, which effect the LU decomposition of the coefficient matrix and the solution of the associated system; tridag.c, which is especially adapted to tridiagonal systems; and svdcmp.c and svbsb.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.

**PYTHON:** The simplest route in PYTHON to find a numerical solution to a system of linear equations was illustrated in Section 5.3.7 and invokes either (1) the command `numpy.linalg.inv`, which returns the inverse of a matrix supplied as its argument and then provides the solution to the equation $Ax = b$ as $x = \text{numpy.linalg.inv}(A) \ast b$ or (2) the command `numpy.linalg.solve(A, b)` which returns the solution to that same equation.

### 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

$$f_1(x_1, x_2) = 0 \quad ; \quad f_2(x_1, x_2) = 0 \quad (14.47)$$

we might begin by drawing a map in the $(x_1, x_2)$ plane showing the zero contours of each function. The map in Fig. 14.14, for example, shows the zero contours for the two functions

$$f_1(x_1, x_2) = \sin\left(\frac{x_1^2}{20}\right) - \cos\left(\frac{x_2}{5}\right) \quad ; \quad f_2(x_1, x_2) = x_1 \tanh(x_2) - 5.0 \quad (14.48)$$

The actual intersections of the dashed and solid curves near the labels A reveal two roots. In addition, the dashed and solid curves pass close to one another—but do not actually intersect—in the vicinity of the labels B. The roots near A can probably be found relatively easily by an iterative search procedure. That there are “almost” roots near B may confuse some algorithms and, if those points are close enough to the real roots near A, they might even cause difficulties in finding the real roots.

As the number of independent variables increases, the search described in the previous paragraph would move from intersection points of curves in a plane to intersection points of three surfaces in three-space to intersection points of four hypersurfaces in four-space to ... 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
CHAPTER 14. FINDING ROOTS

Figure 14.14: Zero contours for the two functions in Eq. (14.48). The function \( f_1(x_1, x_2) \) is shown with solid lines; \( f_2(x_1, x_2) \) is shown with dashed lines. The dashed curve actually intersects the solid curve in two points near the labels A; the two curves almost intersect near the labels B.

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

\[
\begin{align*}
  f_1(x_1, x_2, x_3) &= 0 ; & f_2(x_1, x_2, x_3) &= 0 ; & f_3(x_1, x_2, x_3) &= 0 \\
\end{align*}
\]  

(14.49)

Suppose, further, that we have examined the equations and determined that there does indeed exist a root in the immediate vicinity of the point \((x_1^{(0)}, x_2^{(0)}, x_3^{(0)})\). We might then suppose that the actual root differs from this guess by a small amount, say,

\[
\begin{align*}
  x_1 &= x_1^{(0)} + \delta x_1 ; & x_2 &= x_2^{(0)} + \delta x_2 ; & x_3 &= x_3^{(0)} + \delta x_3 \\
\end{align*}
\]  

(14.50)
and demand that the “corrections” satisfy the equations

\[ 0 = f_1(x_1^{(0)} + \delta x_1, x_2^{(0)} + \delta x_2, x_3^{(0)} + \delta x_3) \]
\[ 0 = f_2(x_1^{(0)} + \delta x_1, x_2^{(0)} + \delta x_2, x_3^{(0)} + \delta x_3) \]  \hspace{1cm} (14.51)
\[ 0 = f_3(x_1^{(0)} + \delta x_1, x_2^{(0)} + \delta x_2, x_3^{(0)} + \delta x_3) \]

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

\[ 0 = f_1 + \frac{\partial f_1}{\partial x_1} \delta x_1 + \frac{\partial f_1}{\partial x_2} \delta x_2 + \frac{\partial f_1}{\partial x_3} \delta x_3 \]
\[ 0 = f_2 + \frac{\partial f_2}{\partial x_1} \delta x_1 + \frac{\partial f_2}{\partial x_2} \delta x_2 + \frac{\partial f_2}{\partial x_3} \delta x_3 \]  \hspace{1cm} (14.52)
\[ 0 = f_3 + \frac{\partial f_3}{\partial x_1} \delta x_1 + \frac{\partial f_3}{\partial x_2} \delta x_2 + \frac{\partial f_3}{\partial x_3} \delta x_3 \]

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.

**MATHEMATICA:** Mathematica’s command `Solve` is capable of solving some sets of nonlinear equations analytically. In that case, the first argument to `Solve` becomes a list containing the equations to be solved and the second argument becomes a list containing the variables for which solutions are sought. When `Solve` can find solutions to a system of nonlinear equations, nesting `Solve` in the command `N` will find numerical solutions. Whether `Solve` is successful or not, numerical solutions to systems of nonlinear equations can be found with `FindRoot`. We simply make its first argument a list of the equations to be solved. Then, for each unknown, we add one further argument in the form of a list containing (first) the name of the variable and (second) an initial guess for the value of that variable in the solution.

**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.

**PYTHON:** The package `scipy.optimize` includes two solvers (`newton_krylov` and `anderson`) for solving large-scale nonlinear systems, two general nonlinear solvers (`broyden1` and `broyden2`), and
three solvers (excitingmixing, linearmixing, and diagbroyden) using simple iterations. Details on these solvers can be found in the PYTHON manuals.\(^\text{40}\)

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

\[
\frac{R(\theta, \alpha)}{v_0^2/g} = \frac{\cos \theta(1 - e^{-\tau})}{\alpha} = \frac{\tau \cos \theta}{\alpha(1 + \alpha \sin \theta)}
\]

\(^{40}\)See particularly the URL docs.scipy.org/doc/scipy/reference/optimize.nonlin.html.
We could view the first of these equations as defining the function $\tau(\theta)$ implicitly. In principle, for a given $\alpha$, we could imagine solving the first equation explicitly for $\tau$ as a function of $\theta$ and substituting that solution into the second equation to find an expression for the range—again for a given $\alpha$—explicitly as a function of $\theta$ alone. If we had that expression in hand, we would find the maximum range by solving the equation $dR(\theta)/d\theta = 0$ for $\theta$ and then evaluate the range at that specific value of $\theta$. In the absence of that expression, we can still differentiate $R(\theta)$ implicitly, recognizing the (hidden) dependence of $\tau$ on $\theta$, and we can differentiate $f(\tau)$. The resulting equations together might then be combinable in a way that would lead to a determination of the maximum range more directly than the route described in the text. Pursue this idea, using a symbolic manipulating program as much as possible to simplify the calculation. The ultimate objective would be to deduce and test a procedure that leads to a numerical value for the maximum range of this projectile when $\alpha$ is given. Note: This exercise is almost certainly difficult and potentially frustrating. The author has no idea whether success awaits the persistent in this endeavor.

14.8. Find the natural frequencies for the three modes of oscillation characterizing the system that results when the system shown in Fig. 14.2 is extended to contain three objects coupled in a line. Take the four springs all to have the same spring constant but allow for the possibility that the middle object may have a mass $m'$ different from the mass $m$ of the two outside objects. In particular, measure frequencies in units of $\sqrt{k/m}$ and seek a graph showing the frequency of each of the modes as a function of $\beta = m'/m$. Hint: To help you get started and to facilitate focusing on the solution of the ODEs rather than on deriving them, note that, for three masses, the equations of motion will be

$$m \frac{d^2x_1}{dt^2} = -kx_1 + k(x_2 - x_1)$$
$$m \frac{d^2x_2}{dt^2} = -k(x_2 - x_1) + k(x_3 - x_2)$$
$$m \frac{d^2x_3}{dt^2} = -k(x_3 - x_2) - kx_3$$

14.9. In dimensionless units, the energy shifts $E$ that occur in the states of hydrogen for $n = 3$ when an external constant electric field is turned on are given by the roots of the ninth-order polynomial

$$f(E) = E^9 - \frac{243}{2} E^7 + \frac{59049}{16} E^5 - \frac{531441}{16} E^3$$

Find the distinct roots of this polynomial and the multiplicity of each root.

14.10. If, when divided by a single line into a square and a rectangle, the resulting smaller rectangle has the same aspect ratio as the original rectangle, the rectangle is called a golden rectangle and the ratio of the longer to the shorter side is called the golden ratio. Rectangles having that ratio are considered to be particularly aesthetic and can be found in many places, including ancient Greek architecture and, intriguingly, in standard, present-day plastic credit cards. Express the above defining criterion to find a quadratic equation for the golden ratio and then solve the equation to find that ratio.

14.17.2 . . . using Numerical Methods

14.11. Use numerical methods to solve Eq. (14.6) for the equilibrium points in the potential energy in Eq. (14.3).

14.12. One way to find the square root of a (positive) number $a$ is to find the root of the function $f(x) = x^2 - a$. (a) Apply Newton's method symbolically to show that $x_{n+1} = (x_n + a/x_n)/2$. (b) Using a pocket calculator and starting with the guess $x_0 = 2$, work out the first few iterates by hand and note how quickly this algorithm converges to $\sqrt{2} = 1.41421$. (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
CHAPTER 14. FINDING ROOTS

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

\[
\tan \alpha = \pm \tanh \alpha
\]


14.15. If the bar of the previous exercise is clamped at one end and free at the other, then the natural frequencies are given by the same expression except that \( \alpha_n \) is instead a solution to the equation

\[
\cot \alpha = \pm \tanh \alpha
\]


14.16. The natural frequencies of the air in a spherical cavity are determined from the roots of the function \( dj_n(x)/dx \), where \( j_n(x) \) is the \( n \)-th order spherical Bessel function, the first three of which are

\[
\begin{align*}
j_0(x) &= \frac{\sin x}{x} \\
j_1(x) &= \frac{\sin x}{x^2} - \frac{\cos x}{x} \\
j_2(x) &= \left( \frac{3}{x^3} - \frac{1}{x} \right) \sin x - \frac{3}{x^2} \cos x
\end{align*}
\]

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.

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\(^{41}\) are 2.4048255577, 5.5200781103, 8.6537279129, 11.7915344391, 14.9309177086, 18.0710639679. Hint: Most computational tools have built-in capabilities for evaluating the Bessel functions. Consult the appropriate vendor manuals.

14.20. Suppose a particle moves in one dimension under the influence of the potential energy

$$V(x) = -\frac{V_0 a^2 (a^2 + x^2)}{8a^4 + x^4} \implies \frac{V(x)}{V_0} = -\frac{(1 + \pi^4)}{8 + \pi^4}$$

where $\pi = x/a$. Using at least three different methods and at least two different computational tools, find the coordinates $\pi$ 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? 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 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. 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 $12x^4 - 12x^2 + 4x - 1$. 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 \texttt{FUNC} and \texttt{FUNCD} to return \( f(x) \) and \( df(x)/dx \), respectively.

- request a tolerance, an initial guess, a maximum number of iterations, and a flag—0 or 1—to be entered when run.
- find the root, terminating iteration either when successive iterates differ by less than the specified tolerance or when the specified maximum number of iterations is exceeded.
- print the final iterate only (flag = 0) or all iterates along the way (flag = 1).
- print a warning if iteration is terminated because the maximum number of iterates was exceeded.
- print the root and the value of the function at that root.

14.26. Write and test a C program paralleling \texttt{bisect.c} but using Newton’s method to find the roots of \( f(x) \). Your program, which you might call \texttt{newton.c}, should

- use the functions \texttt{FUNC} and \texttt{FUNCD} to return \( f(x) \) and \( df(x)/dx \), respectively.
- request a tolerance, an initial guess, a maximum number of iterations, and a flag—0 or 1—to be entered when run.
- find the root, terminating iteration either when successive iterates differ by less than the specified tolerance or when the specified maximum number of iterations is exceeded.
- print the final iterate only (flag = 0) or all iterates along the way (flag = 1).
- print a warning if iteration is terminated because the maximum number of iterates was exceeded.
- print the root and the value of the function at that root.

14.27. The image of a distant object produced on a viewing screen by a small aperture is actually a diffraction pattern. When the aperture is a circle of 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

\[
\frac{I(\theta)}{I_0} = \left[ \frac{2J_1(\pi d \sin \theta/\lambda)}{\pi d \sin \theta/\lambda} \right]^2
\]

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

\[
V_{LJ} = 4\epsilon \left[ \frac{\sigma^{12}}{r^{12}} - \frac{\sigma^6}{r^6} \right] \quad \text{or} \quad \frac{V_{LJ}}{\epsilon} = 4 \left[ \frac{1}{\tilde{r}^{12}} - \frac{1}{\tilde{r}^6} \right]
\]

where \( \epsilon \) and \( \sigma \) are constants and \( \tilde{r} = r/\sigma \), 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 $R_{\text{eff}} = V/I$ for each circuit. Assume that all batteries and resistors have known values and that
quantities represented in the figures with the same symbol have the same value. Warning: For even
simple circuits, Kirchhoff’s laws provide more equations than unknowns. Correctly written, these
equations are guaranteed to be consistent. The subset to be solved, however, must be carefully
chosen to make sure its members are linearly independent of one another.

14.36. The file \$HEAD/data/freefall.dat contains 31 lines, the $i$-th of which contains one numerical
value—the value of the position $x_i$ in cm of a particle at time $t_i = (i - 1)/60$ s. You have reason to
believe that the set of data $(t_i, x_i)$ with $i = 1, 2, 3, \ldots, n$ is described by the parabolic relationship

$$x = at^2 + bt + c$$

The method of least squares identifies the optimum values of the coefficients $a$, $b$, and $c$ as the
particular values that minimize the residual

$$R(a, b, c) = \sum_{i=1}^{n} \left( x_i - (at_i^2 + bt_i + c) \right)^2$$

Using whatever language you choose, write a program that reads the positions from the file into
a vector, generates a second vector containing the corresponding times and then enters a loop in
which it asks for entry of trial values of $a$, $b$, and $c$, calculates and displays $R(a, b, c)$, and returns to
ask for a new set of trial values. Using this program, conduct a manual search for the values of $a$, $b$, and $c$
which make $R(a, b, c)$ as small as possible. Hint: Think carefully about the initial guesses
for $a$, $b$, and $c$. A graph of $x$ versus $t$ may be useful.

14.37. The file \$HEAD/data/freefall.dat contains 31 lines, the $i$-th of which contains one numerical
value—the value of the position $x_i$ in cm of a particle at time $t_i = (i - 1)/60$ s. You have reason to
believe that the set of data \((t_i, x_i)\) with \(i = 1, 2, 3, \ldots, n\) is described by the parabolic relationship
\[ x = at^2 + bt + c \]
The method of least squares identifies the optimum values of the coefficients \(a, b,\) and \(c\) as the particular values that minimize the residual
\[ R(a, b, c) = \sum_{i=1}^{n} \left( x_i - (a t_i^2 + b t_i + c) \right)^2 \]
i.e., as the particular values satisfying the three equations
\[ \frac{\partial R}{\partial a} = 0 \quad ; \quad \frac{\partial R}{\partial b} = 0 \quad ; \quad \frac{\partial R}{\partial c} = 0 \]
These equations will turn out to be linear in \(a, b,\) and \(c,\) with coefficients and inhomogeneities 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 inhomogeneities for the data in the file \$HEAD/data/freefall.dat\ and use at least two different numerical approaches to find the solution for \(a, b,\) and \(c.\) Finally, generate a graph in which each measured point is represented by a simple symbol and the least squares parabola is shown by a solid line so you can judge the adequacy of your fit.

**14.38.** A particle of mass \(m\) moves along the \(x\) axis under the action of a time-dependent force \(F(t).\) We observe that \(x(0) = 0\) and \(x(t_f) = a.\) The detailed motion therefore is described by the solution to the boundary value problem
\[ m \frac{d^2 x}{dt^2} = F(t) \quad , \quad x(0) = 0 \quad , \quad x(t_f) = a \]
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
\[ x_0 = 0 \quad , \quad x_n = a \]
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) \(t_f = 1\ s, a = 10\ m, F(t)/m = e^{-t}\ N/kg\) (exponentially decaying force; \(t\) in seconds), and \(n = 10;\) and
(d) same as (c) but with \(n = 100.\)
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.,

\[
V_{12} = \frac{1}{4} (V_{11} + V_7 + V_{13} + V_{17}) ; \quad V_{15} = \frac{1}{4} (V_{14} + V_{10} + 0.0 + V_{20})
\]

(a) Write out the twenty-five equations implied by this requirement, (b) cast them in the matrix form

\[
\begin{pmatrix}
? & ? & ? & \ldots & ? \\
? & ? & ? & \ldots & ? \\
? & ? & ? & \ldots & ? \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
? & ? & ? & \ldots & ? \\
\end{pmatrix}
\begin{pmatrix}
V_1 \\
V_2 \\
V_3 \\
\vdots \\
V_{25} \\
\end{pmatrix}
= 
\begin{pmatrix}
? \\
? \\
? \\
\vdots \\
? \\
\end{pmatrix}
\]

(figuring out the value represented by each question mark, and (c) solve the system for the unknowns. Then (d) structure the boundary values and solution in a 7 \( \times \) 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

\texttt{\$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).
LISTING OF BISECT.F

CHAPTER 14. FINDING ROOTS

14.A 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.B Listing of \texttt{bisect.c}

/* PROGRAM bisect.c 

This C program asks for the lower and upper bounds of an interval in which a single root of a function is known to exist and then invokes the method of bisection to converge on that root. Refinement of the interval continues until the tolerance TOL---set early in the main program---is reached. The function can be changed by editing its definition in the first lines of code.
*/

#include <stdio.h> /* Load standard i.o routines */
#include <math.h> /* Load standard math routines */

/****** DEFINE FUNCTION ******/

float func( float x )
{
    return pow(x,3)/10000.0 + pow(x,2)/200.0 - x/500.0 - 0.5;
}

void main()
{
    float tol, xa, xm, xb; /* For tolerance, for bounds */
    float fa, fm, fb; /* For values of function */

    tol = 0.0001;

    /****** READ BRACKET VALUES ******/

    printf( "\nLower bound = " ); scanf( "\f", &xa );
    printf( "\nUpper bound = " ); scanf( "\f", &xb );

    /****** EVALUATE FUNCTION AT EXTREMES ******/

    fa = func(xa); fb = func(xb); fm = fa;
    if (fa*fb >= 0.0) { printf( "No root in interval\n" ); return; };

    /****** HUNT FOR ROOT ******/

    while( (xb-xa) >= tol & fm != 0.0 )
    { 
        xm = 0.5*(xb+xa);
        fm = func(xm);
        if (fa*fm >= 0.0) xa = xm; else xb = xm;
    }

    /****** OUTPUT RESULTS ******/

    printf( "root = \%15.6f\n", xm );
    printf( "xb - xa = \%15.6f\n", xb-xa );
    printf( "function = \%15.6f\n", func( xm ) );
}
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 $\texttt{HEAD/tex}$, where (as defined in the Local Guide) $\texttt{HEAD}$ 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 $\TeX$\textsuperscript{1} 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, $\LaTeX$\textsuperscript{2} 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,\textsuperscript{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,\textsuperscript{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.\textsuperscript{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.\textsuperscript{6}

\begin{footnotesize}
\begin{enumerate}
\item As per Knuth’s instruction (see the first two paragraphs of Chapter 1 of \textit{The $\TeX$book} as identified in Section A.18), $\TeX$ is pronounced ‘tech’ as in ‘technology’.
\item 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.
\item 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), \texttt{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.
\item What you see is what you get.
\item See Section A.18 for more detailed references.
\item You may also find the web pages of TUG at the URL \texttt{http://www.tug.org} to be valuable. One link on that page points to an engaging description of the history of the development of $\TeX$.
\end{enumerate}
\end{footnotesize}
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 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 (xemacs, notepad, wordpad, winedt, gedit, ...) and which contains both the text of the desired document and embedded formatting commands. All of \LaTeX{}’s commands begin with a backslash (\). \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 any \LaTeX{} document specifies the document class and has the form\footnote{Be aware that some of the features described in this Appendix are specific to \LaTeX{}2ε and will not work with the previous version (\LaTeX{} 2.09). \LaTeX{}2ε, 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} \ 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 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}
Table A.1: Document classes valid in standard \texttt{L\LaTeX}. Other available classes may be described in the \textit{Local Guide}.

\begin{itemize}
\item \texttt{article} 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 \textit{The \LaTeX\ Manual}.
\item \texttt{report} This class is generally used for longer documents. See Sections 2.2.2 and C.5.1 in \textit{The \LaTeX\ Manual}.
\item \texttt{book} This class is meant for actual books. See Sections 5.1 and C.5.1 in \textit{The \LaTeX\ Manual}.
\item \texttt{slides} 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 \textit{The \LaTeX\ Manual}.
\item \texttt{letter} This class facilitates creating letters. See Sections 5.3 and C.5.1 in \textit{The \LaTeX\ Manual}.
\end{itemize}

Table A.2: The most common options for the \texttt{documentclass} command. Numerous other options are described in Section C.5.1 in \textit{The \LaTeX\ Manual} and in the other references listed in Section A.18.

\begin{itemize}
\item \texttt{10pt} or \texttt{11pt} or \texttt{12pt} (default = 10pt) Specifies default type size.
\item \texttt{letterpaper} or \texttt{legalpaper} or \texttt{a4paper} or \ldots (default depends on site but is usually \texttt{letterpaper} in the US) Specifies size of paper to be used.
\item \texttt{landscape} (default is portrait orientation) Specifies that text is to be formatted for landscape orientation on the specified paper size.
\item \texttt{oneside} or \texttt{twoside} (default = \texttt{oneside}) Specifies whether output is to be formatted for single- or double-sided printing. The \texttt{twoside} option allows for left and right margins and running head positions to be different on odd and even numbered pages.
\item \texttt{onecolumn} or \texttt{twocolumn} (default = \texttt{onecolumn}) Specifies one- or two-column formatting on each page.
\end{itemize}

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 \textit{mandatory} in all documents. Supplemented by the additional feature that a blank line in the code triggers a new paragraph, they are also \textit{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

   \texttt{latex \textit{filename}}

   (Default extension for the input file is \texttt{.tex}).\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 \texttt{.dvi}, which contains the translation of the original
   file into \TeX{}'s device independent language; (2) an ASCII log file, with extension \texttt{.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 \texttt{.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 \texttt{.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 \texttt{.aux} file and a \textit{second}—and occasionally a \textit{third}—pass is necessary to incorporate that
   information in the final formatting of the pages.} The resulting \texttt{.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 \texttt{.dvi} file into a file that can be understood by the printer
   on which the document is to be printed. Most often, the \texttt{.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 dvips. The necessary translation will be achieved either (1) by typing a command like\textsuperscript{12,13}

\texttt{dvips -o filename.ps -t letter filename} \quad (Default extension for the input file is .dvi.)

which translates the entire file into PostScript, stores it in a file, and (with the -o option here used) gives the stored file the same name as the input file but with extension .ps and with the -t option forces use of letter size paper\textsuperscript{14} or (2) by selecting an item from a menu.\textsuperscript{15}

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 .dvi and .ps files with statements like

\begin{verbatim}
l latex texsample1
d vips -o texsample1.ps -t letter texsample1
\end{verbatim}

or equivalent selections from a menu.

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\textsuperscript{16}

\begin{verbatim}
pdflatex filename \quad (Default file type is .tex)
\end{verbatim}

or, for the sample above,

\begin{verbatim}
pdflatex texsample1
\end{verbatim}

\textsuperscript{12}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 Shell prompt or, in the UNIX and LINUX operating systems, the statement \texttt{man dvips} at a Shell prompt.

\textsuperscript{13}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.

\textsuperscript{14}This option is often the default. The option \texttt{-t landscape} will override the default portrait orientation.

\textsuperscript{15}Check the 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}.

\textsuperscript{16}The program \texttt{pdflatex} is normally installed automatically whenever \LaTeX{} is installed.
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 \texttt{latex} works its magic.

2. Starting with the .ps file produced in Section A.1.2, one executes the single statement

\begin{verbatim}
ps2pdf filename.ps
\end{verbatim}

or, for the sample above

\begin{verbatim}
ps2pdf texsample1.ps
\end{verbatim}

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 \texttt{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} (\textit{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 \textit{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 \textit{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 \textit{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 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 The \TeX Manual.

\textwidth (default 4.75”) Specifies the horizontal dimension of the region of the page occupied by text. Its value is commonly 6” for an 8.5”×11” page.

\textwidth (default 0.75”) 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” plus the specified value. The value 0.25” will center 6” wide text on an 8.5”×11” page.

\evensidemargin (default 0.75”) 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” plus the specified value. The value 0.25” will center 6” wide text on an 8.5”×11” page.

\topmargin (default 0.25”) 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” plus the specified value. By default, the height of the header (specified by \headheight) and the separation of the header from the main text (specified by \headsep) together add to 0.5”, so—if the defaults for \headheight and \headsep are accepted—we can pretend that \topmargin specifies the distance from the top edge of the page to the top edge of “real” text to be 1.5” plus the specified value. Thus, for example, the value −0.5” will center 9” high text on an 8.5”×11” page and place the header line—if any—0.5” above the first line of text.

\parindent (default 15.0 points) Specifies the paragraph indent.

\parskip (default 0.0 points) Specifies the extra space between paragraphs.

---

Table A.6: Templates in the directory $\HEAD/tex$.

<table>
<thead>
<tr>
<th>Template</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lu_article.template</td>
<td>Specifies a general article style, including a 6”×9” text area centered on an 8.5”×11” page.</td>
</tr>
<tr>
<td>lu_cpl.template</td>
<td>Specifies the style of CPL publications.</td>
</tr>
<tr>
<td>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>lu_letter.template</td>
<td>Supplies a starting format for business letters.</td>
</tr>
<tr>
<td>lu_memo.template</td>
<td>Supplies a starting format for memos.</td>
</tr>
</tbody>
</table>
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. 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*, 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.

---

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>\TeX{}</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>\emph{text}</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.
Note also in Table A.7 that emphasized text can be achieved not only with the declaration \em 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 \TeX{} 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, \texttt{{\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. \TeX{} 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\{\ldots\}, \textbf\{\ldots\}

to specify medium and bold series, respectively,

\rmfamily, \sffamily, \ttfamily, \textrm\{\ldots\}, \textsf\{\ldots\}, \texttt\{\ldots\}

to specify Roman, sans serif, and typewriter families, respectively, and

\upshape, \itshape, \slshape, \scshape ,
\textup\{\ldots\}, \textit\{\ldots\}, \textsl\{\ldots\}, \textsc\{\ldots\}

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.
A.3. IN-TEXT SPECIFICATION OF LOCAL STYLE

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{text}}</td>
<td>Sets text in tiny type.</td>
<td>Sample</td>
</tr>
<tr>
<td>{\scriptsize \text{text}}</td>
<td>Sets text in scriptsize type.</td>
<td>Sample</td>
</tr>
<tr>
<td>{\footnotesize \text{text}}</td>
<td>Sets text in footnotesize type.</td>
<td>Sample</td>
</tr>
<tr>
<td>{\small \text{text}}</td>
<td>Sets text in small type.</td>
<td>Sample</td>
</tr>
<tr>
<td>{\normalsize \text{text}}</td>
<td>Sets text in normalsize type.</td>
<td>Sample</td>
</tr>
<tr>
<td>{\large \text{text}}</td>
<td>Sets text in large type.</td>
<td>Sample</td>
</tr>
<tr>
<td>{\Large \text{text}}</td>
<td>Sets text in LARGE type.</td>
<td>Sample</td>
</tr>
<tr>
<td>{\huge \text{text}}</td>
<td>Sets text in huge type.</td>
<td>Sample</td>
</tr>
<tr>
<td>{\Huge \text{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 \text{x}, and an em space is about the width of the letter \text{M}, both in the current font.)

- The command \vspace{Value} to generate vertical space. Here, \text{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, \text{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
\begin{verbatim}
\newlength{dblast}
\settowidth{dblast}{**}
\end{verbatim}

and then insert the space for two asterisks (without actually displaying them) with the command
\begin{verbatim}
\hspace{dblast}
\end{verbatim}

These commands are described in detail at the end of Sections 6.4.1 and C.13.1 in \textit{The \TeX\ 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. \TeX\'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{verbatim}
\begin{quotation}
Text of quotation.
\end{quotation}
or
\begin{quote}
Text of quotation.
\end{quote}
\end{verbatim}

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 no extra space between paragraphs while the second will place extra space between paragraphs and will 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{verbatim}
\begin{center}
First line \texttt{\textbackslash} Second line \texttt{\textbackslash} Third line \texttt{\textbackslash} \ldots
\end{center}
\end{verbatim}

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{\textbackslash}, 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{verbatim}
\begin{EnvironmentName}
\ldots
\end{EnvironmentName}
\end{verbatim}

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
A.3. \LaTeX\ Manual, 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 \texttt{\baselineskip} interacts in interesting ways with the formatting of text contained within environments. Changing \texttt{\baselineskip} will affect only the spacing in the main text of the document, including text in \texttt{itemize}, \texttt{enumerate}, and \texttt{verbatim} environments. Footnotes, material in \texttt{tabular} environments, table and figure captions, and perhaps other components will remain single spaced unless—as described in Section C.3.2 in \textit{The \LaTeX\ Manual}—the parameter \texttt{\baselinestretch} is also changed with the command \texttt{\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. \textit{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 \textit{packages}, the features of which can be made available in any specific code by placing the command

\begin{verbatim}
\usepackage{ PackageName }
\end{verbatim}

in the preamble \textit{The \LaTeX\ Companion} describes numerous packages, including \texttt{amstex}, \texttt{b babel}, \texttt{color}, \texttt{graphics}, \texttt{graphicsx}, \texttt{graphpap}, \texttt{ifthen}, \texttt{latexsym}, \texttt{imakeidx}, \texttt{verbatim}, \texttt{pict2e}, and \texttt{showidx}. A brief description of many of these packages is included in Section C.5.2 of \textit{The \LaTeX\ Manual}. The statement \texttt{texdoc PackageName} at a \textit{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 \texttt{~} instead of a space between the words. The tilde should also be used after a period that does \textit{not} end a sentence. For example, we should type \texttt{‘Mr.~Cook’} rather than \texttt{‘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 \texttt{--}, \texttt{---}, or \texttt{---}, 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 dash.

\footnote{The symbol ’\_’ underscores the presence of a space at the indicated point; it is \textit{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 \ at the point in a word where \TEX is permitted to insert a hyphen as it fills and justifies lines. \TEX has its own rules for hyphenation, but they are not infallible. Sometimes \TEX 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 command, e.g., \mbox{customize}.

### A.4 Including Equations

Most scientific documents will have equations, all of which must be specified in \TEX’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 \{\ldots\} or \$\ldots\$, 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\ldots\end. 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 \TeX 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, \TEX 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.\footnote{When 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.} (Warning: The displaymath, equation, and eqnarray environments will generate error messages if they contain blank lines.)
including lists

\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 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., $\\Theta$ for $\Theta$ and $\omega$ 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., $\kappa$, 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 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 $\cos$ and $\sin$ should be used for these functions.\footnote{See Table 3.9 in \textit{The \LaTeX Manual} for a complete listing.} Except for $\sc$, the declarations shown in Table A.7 can also be used in math mode to specify text in something other than italic 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 two-word phrase, will result in the words running together unless the 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


dollar$\cdots \texttt{mbox{\texttt{boldmath}$\theta$}} \cdots \dollar$

dollar

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{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 $\texttt{\begin \cdots \texttt{\end}}$. Within the environment, each new item is introduced with the command $\texttt{item}$. Thus, the structure might look like

\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}.}
The bullets and the numbers are, of course, generated automatically and, within the `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 \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 \item[DMC] will cause the item to be labeled `DMC' and the command \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 \item{} to prevent the text in square brackets from being interpreted as the desired label.

- The length parameters \itemsep, \parskip, and \parsep, which provide a means to override default spacings within the environment.

- The way to change the “bullets” in the itemize environment. The symbols used to label the “bulleted” items in the various levels of nested itemize environments are created by the commands \labelitemi, \labelitemii, \labelitemiii, and \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 itemize environments.

- The way to change the form of the labels for each item in the enumerate environment. The labels at the various levels of nested enumerate environments are determined by invoking one of the commands \theenumi, \theenumii, \theenumiii, and \theenumiv, whose action is in turn determined from the value of the corresponding one of the counters enumi, enumii, enumiii, and 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}{$(\textbf{\Alph{enumi}})$}
\end{verbatim}

which will change the labels used at the first level in enumerate environments to upper-case arabic letters in parentheses. The command \Alph could be replaced with \alph, \roman, \Roman, or \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 tabular environment, the details of which are involved and are fully described in *The \TeX Manual*. The 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,\textsuperscript{31} and \LaTeX{} provides the \texttt{table} environment to facilitate proper placement of a table,\textsuperscript{32} 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.\textsuperscript{33} 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.\textsuperscript{34,35}

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 \textit{above} the table. Positioning the commands \texttt{\caption} and \texttt{\label} after the \texttt{center} environment would place the caption \textit{below} the table. Whichever position is adopted, the command \texttt{\label} must appear \textit{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.

\begin{itemize}
\item \textsuperscript{31}Actually, the command \texttt{\begin{table}} has an optional argument \texttt{\begin{table}[OptArg]}, which can assume any of the values \texttt{h}, \texttt{b}, or \texttt{t} for placement of the table at the place where the \texttt{table} environment appears (\texttt{h}, for here) or at the bottom (\texttt{b}) or top (\texttt{t}) of the page. In the absence of an explicit specification, \LaTeX{} makes its own decision about placement.
\item \textsuperscript{32}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.
\item \textsuperscript{33}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.
\item \textsuperscript{34}\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 The \texttt{\LaTeX} Manual for details.
\item \textsuperscript{35}See Sections 3.6 and C.10 in The \texttt{\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.
\end{itemize}
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\textsuperscript{36}

- 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 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.\textsuperscript{37} 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,

\textsuperscript{36} 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 The \LaTeX\ Graphics Companion.

\textsuperscript{37} 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{} (not \LaTeX) 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 \texttt{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.\footnote{More generally, this statement admits an optional argument to the desired graphics driver explicitly. Frequently, that driver will be \texttt{dvips} and the statement would be \texttt{\usepackage[dvips]{graphicx}}, but numerous other drivers exist, and many may be installed at your site. See the \textit{Local Guide} for more information. For ultimate flexibility in the formatting of graphics files, we elect to leave that choice to the default.}

After placing the command \texttt{\usepackage\{graphicx\}} in our preamble, we incorporate the graphical display itself by placing a statement like

\begin{verbatim}
\includegraphics[height=\texttt{dimension}]\{\texttt{filename}\}
\end{verbatim}

at the point where the PostScript or PDF figure defined in by the file \texttt{filename} is to be inserted.\footnote{If you use \texttt{latex}, \texttt{dvips}, and \texttt{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 \texttt{\includegraphics} command—the processing will correctly identify and incorporate the figures. If you seek to produce a PDF document directly with \texttt{pdflatex} as described in item 1 in Section A.1.3, then—even if the file type .pdf is omitted in the \texttt{\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 \textit{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 \texttt{\includegraphics} command, then you can use either \texttt{latex} or \texttt{pdflatex} for the processing.}

In general, the image would be centered horizontally on the page by inserting this command in a \texttt{center} environment, e.g.,

\begin{verbatim}
\begin{center}
\includegraphics[height=\texttt{dimension}]\{\texttt{filename}\}
\end{center}
\end{verbatim}

For example, the code\footnote{Appropriately translated, the substance of footnote 34 applies here to the entry placed automatically 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.\textsuperscript{43,44} 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 \textit{before} the command \texttt{\caption}, so that the caption will appear \textit{above} the figure. Further, the command \texttt{\label}, which must \textit{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

\textsuperscript{43}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.

\textsuperscript{44}The label following the word ‘Figure’ in the output will, of course, reflect the document class in use and the unit of the document in which the figure appears.
the aspect ratio. If both height and width are specified, each will be respected and the aspect ratio
of the figure may be distorted; specification of 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 The \BTeX Graphics
Companion.

A.7.3 Using the tikz Package

The \texttt{tikz} package, which is routinely included in present-day \LaTeX distributions, provides an assort-
ment 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 not contain the correct figures.} 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} 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.}
texdoc pgf

at a \textit{Shell} window to your operating system will probably bring up links to a number of manu-
als, 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 *\LaTeX* package `color`.}

```
definecolor{ColorName}{rgb}{r,g,b}
```

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 *\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 *π* 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

```
\foreach x in {0.0,1.0,2.0} {
  \draw (x,0.1 )--(x,-0.1);
  \node at (x, -0.4){x};;
}
```

creates a loop executing the statements enclosed in `{...}` for each value in the list following the `\foreach` 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 *\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.

\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) {\textbf{y}};
\node at (2.4,0.0) {\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} \hspace{1in} (Default file type .tex)

will produce a PDF file and the statements

\texttt{latex filename} \hspace{1in} (Default file type .tex)
\texttt{dvips -o filename.ps -t letter filename} \hspace{1in} (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 \texttt{latex} or \texttt{pdflatex} will be necessary. If there are PostScript figures and no PDF figures to be incorporated, only \texttt{latex} will work; if there are PDF figures (and no PostScript figures) or hyperlinks to be incorporated, only \texttt{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 \texttt{tikz} code, you might wish to explore a WYSIWYG editor \texttt{TikzEdt} that provides a graphical interface for creating figures and translates that figure into the corresponding \texttt{tikz} code. This program, which is free, can be downloaded from the web site \url{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 \texttt{picture} Environment and \texttt{pict2e} Package

Finally, we merely mention the \texttt{picture} environment, described in Sections 7.1 and C.14.1 of \textit{The \LaTeX{} Manual}, and the supplementary \texttt{pict2e} package, described in \textit{The \LaTeX{} 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 \texttt{\tableofcontents{}}, \texttt{\listoffigures{}}, and \texttt{\listoftables{}}. Each

- results in the output of an auxiliary ASCII file of information with file type \texttt{.toc}, \texttt{.lof}, and \texttt{.lot}, respectively,
- requires a second pass through \texttt{latex} or \texttt{pdflatex} to incorporate the list in the finished document, and
- 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 \texttt{\part{}}, \texttt{\chapter{}}, \texttt{\section{}}, \texttt{\subsection{}} and \texttt{\subsubsection{}} will automatically generate a line in the \texttt{.toc} file;\footnote{Note that not all of these options are available in all document classes. For example, \texttt{\part{}} and \texttt{\chapter{}} are available only in the \texttt{book} class. Note also the counters \texttt{secnumdepth} and \texttt{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 \texttt{\setcounter{}} in the preamble.} each use of the \texttt{\caption} command in a \texttt{figure} or \texttt{table} environment will automatically generate a line in the \texttt{.lof} or \texttt{.lot} file. Then, in the second pass—which will always be necessary—through \texttt{latex} or \texttt{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 \LaTeX{} source file, specifically

- the command \texttt{\addcontentsline{\textit{file}}{\textit{unit}}{\textit{entry}}}
- the command \texttt{\addtocontents{\textit{file}}{\textit{text}}}
A.9. INCLUDING AN INDEX

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 \\texttt{\vspace{24pt}This is a note.\vspace{24pt}} and the several sectioning commands (\texttt{\chapter}, \texttt{\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
\printindex
```

in the preamble and the command

```
\index{Item to be indexed}
```

at appropriate points throughout the document. (The detailed structure of the argument to the command \texttt{\index} is described in Section 4.5 and Appendix A of The \LaTeX Manual.) When the file \texttt{filename.tex} is processed by \texttt{latex} or \texttt{pdflatex}, the presence of these commands results in the addition of three auxiliary ASCII files named

- \texttt{filename.idx} containing one line for each index entry,
- \texttt{filename.ilg} containing a log of messages created when, behind the scenes, the idx file is automatically processed through \texttt{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 \texttt{makeidx}, which is still available. We here recommend using \texttt{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.}
• `filename.ind` produced by `makeindex` and containing the formatted index that is then read by \texttt{\textbackslash printindex} to create the actual index in the document.

With \texttt{imakeidx}, the index is automatically created and incorporated in the document because an execution of `makeindex` is inserted automatically at the appropriate point when \texttt{latex} or \texttt{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).

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 \textit{The \LaTeX{} Companion} as identified in Section A.18.

To assist in constructing the index, an auxiliary package called \texttt{showidx}, which works both with \texttt{latex} and with \texttt{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 (\texttt{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 \texttt{ghostview} or acroread. Conveniently, if the .dvi file is displayed on the screen with \texttt{xdvi} (UNIX) or \texttt{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 `\texttt{\textbackslash marginparwidth}` (which sets the width of the marginal boxes) and `\texttt{\textbackslash 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.\footnote{With \texttt{makeidx}, a separate explicit processing of the .idx file with \texttt{makeindex} to produce the .ind file and a further pass through \texttt{latex} or \texttt{pdflatex} was necessary to include the index in the document.}

\footnote{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 \TeX 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 \TeX 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.\footnote{The default value of the \texttt{bookmarksnumbered} and \texttt{linktocpage} options is \texttt{false}.} 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.\footnote{See item 25 in Section A.17 for ways to achieve this deletion.}

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\footnote{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}.}
APPENDIX A. INTRODUCTION TO \LaTeX

• (after first pass through \texttt{pdflatex}) \texttt{.aux}, \texttt{.idx}, \texttt{.ilg}, \texttt{.ind}, \texttt{.log}, \texttt{.out}, \texttt{.pdf}, and \texttt{.toc}. The \texttt{.idx}, \texttt{.ilg}, and \texttt{.ind} files will be present only if you are creating an index with \texttt{imakeidx}. The \texttt{.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 \texttt{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 \textit{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 \textit{above} the item rather than below the item.

A \textit{CONVENIENCE}: If the commands \texttt{\textbackslash documentclass} and \texttt{\textbackslash usepackage\{graphicx\}} are phrased \textit{without} options \textit{and} the files in all \texttt{\textbackslash includegraphics} commands are presented \textit{without} file type, then the file processed with \texttt{latex} will look for \texttt{.ps} and \texttt{.eps} graphics files and the file processed with \texttt{pdflatex} will look for \texttt{.pdf} files. All files to be sought must, of course, exist.\footnote{See Section A.11 for a means to convert \texttt{.ps} and \texttt{.eps} files to \texttt{.pdf} and \textit{vice versa}.} This feature makes it easy to process the \textit{same} source file both with \texttt{latex} and with \texttt{pdflatex}.

A.11 Converting \texttt{.eps} and \texttt{.ps} Files to \texttt{.pdf}

When files describing figures are created, it is often easier to output those files from the creating program as \texttt{.eps} or \texttt{.ps} files rather than as \texttt{.pdf} files. Unfortunately, if you desire to produce the final output with \texttt{pdflatex}, files describing pictures need to be \texttt{.pdf}. PostScript files must therefore be converted into PDF before running \texttt{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 \texttt{figure.eps} and \texttt{figure.ps} to a useful PDF file involves the two steps

1. \texttt{ps2pdf figure.eps} \hspace{1cm} (which will yield \texttt{figure.eps.pdf})
   \texttt{ps2pdf figure.ps} \hspace{1cm} (which will yield \texttt{figure.pdf})

2. \texttt{pdfcrop figure.eps.pdf} \hspace{1cm} (which will yield \texttt{figure.eps-crop.pdf} or \texttt{figure.pdf-crop.pdf})
   \texttt{pdfcrop figure.pdf}

Here, step 1 creates the \texttt{.pdf} file and step 2 removes extraneous white space around the perimeter of the first-created \texttt{.pdf} file. Second arguments to both \texttt{ps2pdf} and \texttt{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 \textit{Shell} scripts listed in Section A.A.2, respectively, exploit the second argument for both \texttt{ps2pdf} and \texttt{pdfcrop} to control the file names and, when executed with statements like

\begin{verbatim}
  MS DOS                 UNIX
  ceps2pdf figure        ./ceps2pdf figure (for .eps file)
  cps2pdf figure         ./cps2pdf figure (for .ps file)
\end{verbatim}
A.11. CONVERTING .EPS AND .PS FILES TO .PDF

will (1) leave in the involved directory a cropped PDF file whose name \texttt{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 \texttt{Command} window (Windows) or a \texttt{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 \texttt{$\text{HEAD}/tex$}.

- Copy all .ps or all .eps files to be converted into that temporary directory. \textit{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

  \begin{align*}
  \text{MS DOS} & \\
  \text{dir} /b *.eps > \text{dir.txt} & \\
  \text{dir} /b *.ps > \text{dir.txt} & \text{ \ \ \ \ UNIX} \\
  \text{ls} -1 *.eps > \text{dir.txt} & \\
  \text{ls} -1 *.ps > \text{dir.txt} \\
  \end{align*}

  utilizing the option /b in Windows and the option -1 (one, not el) in UNIX produce the desired file (filename, including file type).

- Strip the file type by running the python program \texttt{ExtractFileName.py} (Section A.A.3). Here the same code works in both Windows and UNIX. This action will create the file \texttt{nameonly.txt} containing only the names of the files to be converted.

- Effect the conversions by running the script \texttt{rdfile} with statements like

  \begin{align*}
  \text{MS DOS} & \\
  \text{rdfileeps} & \text{(for eps files)} \\
  \text{rdfileps} & \text{(for ps files)} \\
  \text{ \ \ \ \ UNIX} \\
  \text{./rdfileeps} & \text{(for eps files)} \\
  \text{./rdfileps} & \text{(for ps files)} \\
  \end{align*}

  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 \texttt{latex} or \texttt{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

\begin{align*}
\text{pdftops} & \text{ FileName.pdf FileName.ps} \\
\text{pdftops} & \text{ -eps FileName.pdf FileName.eps}
\end{align*}

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

```latex
\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

```latex
\newboolean{FlagName}
\setboolean{FlagName}{FlagValue}
```

where \textit{FlagName} can be any name that does not conflict with names already used and \textit{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

```latex
\ifthenelse{\boolean{FlagName}}{\TextToBeInsertedIfFlagValueIsTrue}{\TextToBeInsertedIfFlagValueIsFalse}
```

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

```latex
\usepackage{ifthen}
```

appears and several flags are defined and set with commands like, for example,

```latex
\newboolean{LATEX} \setboolean{LATEX}{true}
```

Finally, the command

```latex
\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{\&}, \texttt{\|}, and \texttt{\neg}. For example, if the Boolean flags \texttt{NUMREC} and \texttt{FORTRAN} are defined and set, then the composite statement

```latex
\ifthenelse{\boolean{NUMREC} \& \boolean{FORTRAN}}{\input{FileName}}{}
```
would include *FileName* only if *both* *NUMREC* and *FORTRAN* are true.

In the effort to render files processable with either \texttt{latex} or \texttt{pdflatex} and also to create easily either a printable document or a linked document, the package \texttt{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\footnote{Inclusion of the package \texttt{ifthen} can be omitted if it had been entered previously for other reasons.}

\begin{verbatim}
\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} }
\end{verbatim}

which define the boolean variable \texttt{PRINT}, ask for entry of either \texttt{true} or \texttt{false} at execution time, and terminate execution if an invalid value is entered. Then, the lines

\begin{verbatim}
\usepackage{color}
\ifthenelse{\boolean{PRINT}}{}{\definecolor{MyPurple}{rgb}{0.577,0.000,1.000}
\usepackage[colorlinks=true, linkcolor={MyPurple}]{hyperref}}
\end{verbatim}

also in the preamble, will include the package \texttt{hyperref}, but only if \texttt{PRINT} is false.

One further conditional command included in \texttt{latex} and \texttt{pdflatex} without adding a package facilitates testing whether a file exists. For example, the statement

\begin{verbatim}
\IfFileExists{test.tex}{\input{test.tex}{}}
\end{verbatim}

inputs \texttt{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 \texttt{latex} or \texttt{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.,

\begin{verbatim}
\IfFileExists{test.tex}{\input{test.tex}{\typeout{File not found} \end{document}}
\end{verbatim}

\section*{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 1 (el) is followed (1) by the number of the line in the code at which the problem is detected
and (2) the text surrounding the problem. Finally, the question mark prompts for user input. A simple ⟨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 x (followed by ⟨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 before a mathematical expression or back to text mode after a mathematical expression. All of the reported errors must be identified and fixed (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 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 Local Guide describes the page previewer available at your site. UNIX systems usually provide a version of xdvi and Windows systems usually provide a version of 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 xdvi filename or yap filename (The default extension for the input file is \texttt{.dvi}.) or select an appropriate item from a menu (as described in The Local Guide). Presently, the first page of the document will appear on the screen. Some versions of xdvi display an array of buttons along the right edge of the Xdvi window; other, probably newer, versions replace those buttons with drop-down menus and icons in a toolbar. Clicking ML on any of the various buttons or selecting items from one of the menus effects the associated action. For 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 xdvi have a window along the left edge in which you can click 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 Xdvi window, 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 xdvi documentation, e.g., the UNIX 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 and a page previewer simultaneously involves invoking the previewer with a command like xdvi filename & (UNIX) or yap filename & (Windows)

---

65 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.  
66 Yet Another Previewer  
67 Additional information about xdvi on UNIX systems can be found in the on-line help, accessed in many systems by typing the command \texttt{man xdvi}.  
68 If explicit specification of the input file is omitted, some implementations of 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 File menu.
Table A.11: Some of the features available through selection from menus (first column) in some versions of xdvi, through clicking ML on buttons in other versions of xdvi. The features actually available by these means may vary from version to version of 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 xdvi.

<table>
<thead>
<tr>
<th>Keystroke</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>nn</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 xdvi or yap but detach it from the launching Shell window, leaving the Shell window free for other uses. Then, with the .tex file in a text editor detached from the launching Shell window, we can edit the text, save the edited version, use the Shell window to invoke \LaTeX{} on the edited file, and then simply click ML at an appropriate point in the displayed text to instruct xdvi or yap to reread the .dvi file. 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

The programs ispell and aspell, either or both of which may be part of your \LaTeX{} distribution, are common spell checkers. If either is installed at your site (see the Local Guide), it can be invoked

\footnote{In some environments, rereading an updated file will happen automatically without an explicit user request.}

\footnote{To the author’s knowledge, no comparable stand-alone spell checker exists for Windows.}
APPENDIX A. INTRODUCTION TO \LaTeX

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} or \texttt{aspell check filename}

or perhaps with a mouse click in a GUI. For files with extension .tex, \texttt{ispell} enters its \TeX{} mode and will not identify every \TeX{} or \LaTeX{} 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 \LaTeX{} is \emph{not} assessed.

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 comments following the \% sign can be included in the code but have no effect on the output produced.)

\documentclass{article} % Mandatory command; select
  % default 10 point type
\setlength{\textheight}{9truein} % Set height of text area
\setlength{\topmargin}{-0.5truein} % Center text on 11'' height
\setlength{\textwidth}{6truein} % Set width of text area
\setlength{\oddsidemargin}{0.25truein} % Center text on 8.5'' width
\setlength{\parskip}{6pt plus 1pt minus 1pt} % Specify extra space
  % between paragraphs,
  % allowing \LaTeX{} to adjust
  % space 1 point up or down
\setlength{\parindent}{40pt} % Set paragraph indent
\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\ldots}: J.~Q.~Student \\ % \\ forces a new line
{\em Date\ldots}: \today \\ % Note italic correction
{Tilde prevents extra space}
\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''\times\!11''
page.\footnote{Note the special command \texttt{$\backslash$t\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}}

\texttt{\begin{flushleft}
{\em Date\ldots}: \today
\end{flushleft}}

\begin{verbatim}
\documentclass{article}
\setlength{\textheight}{9truein} % Set height of text area
\setlength{\topmargin}{-0.5truein} % Center text on 11'' height
\setlength{\textwidth}{6truein} % Set width of text area
\setlength{\oddsidemargin}{0.25truein} % Center text on 8.5'' width
\setlength{\parskip}{6pt plus 1pt minus 1pt} % Specify extra space
  % between paragraphs,
  % allowing \LaTeX{} to adjust
  % space 1 point up or down
\setlength{\parindent}{40pt} % Set paragraph indent
\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\ldots}: J.~Q.~Student \\ % \\ forces a new line
% Note italic correction
{Tilde prevents extra space}
\end{flushleft}
\end{verbatim}
command---i.e., in the preamble---followed by the command \verb*+\maketitle+ after the beginning of the document. The command \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 \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} \, 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 $dx$ in Eq.~\ref{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 \texttt{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 \texttt{$HEAD/tex$}.

\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 \texttt{\today}, which returns today's date in the form “month date, year”.
4. The command \texttt{\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 1. Further, the page number that would appear has been deleted in this display.

A Sample Document for Your Perusal


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" x 11" page.\textsuperscript{a} The title could have been generated using the three \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*+ \ldots \ldots +} 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

$$
\int_0^\infty e^{-x} \, dx = 1,
$$

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

$$
\frac{\partial e^{x^2 y^2}}{\partial x_0} = y^2 e^{x^2 y^2}
$$

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}.

\begin{footnotesize}
\textsuperscript{a}Note the special command \texttt{\LaTeX} provided for the display of the \LaTeX “logo”.
\end{footnotesize}

\begin{verbatim}
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

\texttt{\renewcommand{\today}{\number\day\space ifcase\month\or January\%}}

\texttt{\or February\or March\or April\or May\or June\or July\or August\%}

\texttt{\or September\or October\or November\or December\fi \space\number\year}

The percent signs at the end of the first two lines are \textit{not} superfluous; they guarantee that \LaTeX will see this command as a single line and hence that extraneous spaces will not appear in
\end{verbatim}
the output for occasional dates. Note also that this modification invokes \TeX's \texttt{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 \textit{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 \textit{The \TeX\ Manual}.

7. The command \texttt{\thispagestyle{Style}}. Placed at any point, this command selects a particular page style \textit{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 \textit{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 \textit{The Rev\TeX\ Input Guide} prepared by the AIP, the APS, and the OAS.\textsuperscript{72}

9. The command \texttt{\input}, which simply inputs the file specified in its argument. Details are described in Section 4.4 of \textit{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 \textit{The \TeX\ Manual}.

11. Methods for placing accents over characters. For example, the command \texttt{"{o}} will produce ö, the command \texttt{"{n}} will produce ñ, and the command \texttt{c{c}} will produce ç. A full listing of the possibilities will be found in Section 3.2.1 and Table 3.1 of \textit{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}
\input{FileName}
\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 \textit{not} be recognized as introducing a command. If, instead, we invoke the alternative \texttt{alltt} environment with the \TeX\ commands

\begin{verbatim}
\begin{alltt} \input{FileName} \end{alltt}
\end{verbatim}

the problem is solved, since \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.

\textsuperscript{72}Version 4.1, which was released in final form on 11 August 2010, is compatible with \TeX\2\epsilon. The APS website \texttt{publish.aps.org/revtex4} (no \texttt{www.}) contains up-to-date information about Re\TeX, links to an assortment of manuals (including one titled \textit{Re\TeX\ Author's Guide}, and a link from which that version can be downloaded. Re\TeX\ is automatically included in many standard \TeX\ distributions.
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 embelishsed with alltt 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 \verbatimend on a line by itself.}

13. The ten characters ($, &, {, ], %, #, \, ~, ^, \), 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 \^ 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^{\}. 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 (not \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 the characters ‘b’, ‘^’, ‘\’, and ‘\’, respectively. Similarly, the command {\tt \char‘>} will produce ‘>’, which differs in sometimes desirable ways from the character ‘>’ produced by $\backslash$.\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 \verbatimend on a line by itself.}

14. The commands that allow us to create boxes as described in Sections 6.4.3 and C.13.3 in The \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

\newcounter{hold}

where \textit{hold}, which cannot contain numeric characters, is the name chosen for the counter. Then, the structure

\begin{enumerate}
\item Text of first item.
\item Text of second item.
\setcounter{hold}{\value{enumi}}
\end{enumerate}

... Text not in enumerate environment.

\begin{enumerate}
\setcounter{enumi}{\value{hold}}
\item Text of third item.
\item Text of fourth item.
\setcounter{hold}{\value{enumi}}
\end{enumerate}

will achieve the desired end. In the first enumerate environment, the counter \textit{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, \textit{enumi} stores the number of the last item. The command \texttt{\setcounter} in the first environment saves the current value of \textit{enumi} in the counter \textit{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 \textit{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

\renewcommand{\thesection}{\Alph{section}.}
\renewcommand{\thesubsection}{\thesection\alph{subsection}.}

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

\renewcommand{\thetable}{\Roman{table}}
\renewcommand{\thepage}{\roman{page}}
Table A.14: Structure to produce full-width text at top of double-column text.

\documentclass[twocolumn]{article}
\begin{document}
\begin{center}
{\large\bf Title} \\
{\large\bf Author} \\
\end{center}
\begin{quotation}
... Text of abstract.
\end{quotation}
\end{center}
\begin{quotation}
... Text of paper
\end{quotation}
\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 ... 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 \textit{same} page as) a two-column presentation of text, which exploits an optional argument to the \texttt{\begin{document}} \texttt{\twocolumn} (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{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}.} 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

\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, \hbox{L}a\hbox{\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 \texttt{\tablename} and can be changed in the way described for \texttt{\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, \hbox{L}a\hbox{\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 \texttt{\chaptername} and can be changed in the way described for \texttt{\figurename} at item 19 above.

22. The means to change the title above the table of contents and above the index. By default, \hbox{L}a\hbox{\LaTeX} labels the table of contents ‘Contents’ and the index ‘Index’. The word ‘Contents’ is supplied by the command \texttt{\contentsname} and the word ‘Index’ is supplied by the command \texttt{\indexname}. Each can be changed in the way described for \texttt{\figurename} at item 19 above.

23. Using the \texttt{array} environment to create matrices in math mode. As with the \texttt{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 \texttt{\} marks the end of each row. Thus, for example, the \hbox{L}a\hbox{\LaTeX} code

\[
\left(\begin{array}{cccc}
1 & 2 & 3 & 4 \\
5 & 6 & 7 & 8 \\
9 & 10 & 11 & 12
\end{array}\right)
\]

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 \hbox{L}a\hbox{\LaTeX} package \texttt{needspace}, which facilitates avoiding awkward page breaks conditionally. For example, when a section title occurs close to the bottom of a page, \hbox{L}a\hbox{\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 \texttt{\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

\texttt{\usepackage{needspace}}

in the preamble and then, at the point where a conditional page break might be wise, inserting one of the commands

\texttt{\needspace{5\baselineskip}} or \texttt{\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 \texttt{\needspace} will depend some on what penalties are in effect but will usually be close enough to be acceptable; further, \texttt{\needspace} will leave a ragged bottom even if \texttt{\flushbottom} is in effect. The command \texttt{\Needspace} will leave the requested space, will take longer to execute, and will leave a ragged bottom; the command \texttt{\Needspace{*}} will produce a flush bottom if \texttt{\flushbottom} is in effect. The unit of measure is the size of the length parameter \texttt{\baselineskip}. The 5 in this example can, of course, be whatever number seems appropriate. Note that there is no multiplication sign after the number.
25. The Windows batch command `cleanTEX` defined by the batch file

```batch
REM 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

```bash
#!/bin/bash
# delete all TEX intermediate files
rm -f *.sav *.out *.ind
```

which facilitate removing all \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\ 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\ auxiliary files.

26. The Windows batch command `cleanbak` define by the the batch file

```batch
REM Delete .bak files
del /s *.bak
del /s *.sav
```

and the UNIX shell script `cleanbak` defined by the file

```bash
#!/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 `\texttt{man \ hyperref}`, e.g. `\texttt{man hyperref}`, may bring up a page describing the indicated entity. In all full \TeX installations, the Shell command `\texttt{texdoc \ hyperref}`, 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 (Birkhäuser, 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 \TeX 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 (\texttt{center} environment);
- a right-justified date (using \texttt{\today} and either \texttt{\hfill} or the \texttt{flushright} environment);
- a left-justified block containing an inside address (\texttt{flushleft} environment);
- a left-justified salutation;
- the body of the letter, containing more than one paragraph;
- a closing (e.g., Sincerely, With love, \ldots) 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 $\$HEAD/tex$ 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. \textit{A Crutch:} The file $\$HEAD/tex/sampledoc.tex$ 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}
MEMORANDUM
\end{center}

Current date

\begin{center}
TO: Insert name of recipient here.
FROM: Insert name of sender here.
SUBJECT: Insert topic here.
\end{center}

MESSAGE:

Insert message here.

Make sure that the date placed in the memo is the date the memo was processed (i.e., use the command \texttt{\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 528. 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>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original grant</td>
<td>$xxx,xxx</td>
<td></td>
</tr>
<tr>
<td>Total expenses reported last time</td>
<td>$xxx,xxx</td>
<td></td>
</tr>
<tr>
<td>Available funds at start of current period</td>
<td>$xxx,xxx</td>
<td></td>
</tr>
</tbody>
</table>

Expenses in current period:

<table>
<thead>
<tr>
<th>Category</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<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
pdfcrop 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
pdfcrop 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
   pdfcrop tmp.pdf $filename.pdf
   rm -f tmp.pdf
done
rm -f dir.txt
rm -f nameonly.txt

Shell Script rdfileps

#!/bin/bash
# rdfileps  (Convert each file in nameonly.txt to PDF in UNIX)
cat nameonly.txt | while read filename
do
   ps2pdf $filename.ps tmp.pdf
   pdfcrop tmp.pdf $filename.pdf
   rm -f tmp.pdf
done
rm -f dir.txt
rm -f nameonly.txt
A.A.3  ... for Windows and UNIX

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 -1
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. Additional information and downloads may be found from links at www.netlib.org/odepack. In particular, the two text files opkd-sum and opks-sum 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.)

**MacTeX**

MacTeX is a shareware implementation of the TeX/LaTeX system for Macintosh computers. Information is available at the URL www.tug.org/mactex.

**MAXIMA**

Originally called MACSYMA, this first of the computer algebra systems was developed at MIT and supported from its origin in 1968 until 1982 by MIT, NASA, ONR, and DOE. In 1982, MACSYMA became a commercial product that was further developed and remained available until about 1999.
The 1982 MIT version remained available as DOE MACSYMA but was released in 1999 to a group that continues to develop and maintain the program, changing its name to MAXIMA, whose website is at the URL http://maxima.sourceforge.net/. MAXIMA is freely available for a wide variety of platforms.

**MAPLE®**

Waterloo Maple, Inc.  
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 T\TeX/L\T\EX system for Windows computers. Information is available at the URL www.miktex.org. MiKTeX is also available from CTAN, the T\TeX/L\T\EX 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 \url{www.trevorrow.com/oztex}. OzTeX is also available from CTAN, the \TeX/LaTeX distribution network accessible from links at \url{www.tug.org}.

PYTHON/NUMPY/MATPLOTLIB/PLOTLY/MAYAVI/SCIKIT-IMAGE

PYTHON, a high-level programming language created by Guido van Rossum, was first released in 1991. In execution, PYTHON programs are interpreted, not compiled. Information and free downloads are available at the URL \url{www.python.org}. Information about various add-on modules can be found as follows:

- The module \texttt{numpy}, which adds numerous mathematical capabilities (arrays, matrices, mathematical functions, ...) to PYTHON, is described at the URL \url{www.numpy.org}.
- The module \texttt{matplotlib}, which adds numerous 2D plotting capabilities and a few 3D plotting capabilities to PYTHON, is described at the URL \url{matplotlib.org}.
- The module \texttt{plotly}, which provides several 3D plotting capabilities, is described at the URL \url{plot.ly}.
- The module \texttt{mayavi}, which provides numerous capabilities for 3D data visualisation and plotting, is described at the URL \url{docs.enthought.com/mayavi/mayavi}.
- The module \texttt{scikit-image}, which includes \texttt{scimage} and provides image processing capabilities, is described at the URL \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 \url{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), \url{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 \url{www.tug.org/texlive}.

TGIF
TGIF (pronounced T-G-I-F) is a versatile program for creating two-dimensional drawings. The program is Xlib-based and interactive, and it runs under X11 on LINUX and UNIX platforms (including MAC OS X and cygwin on Windows). Information (brief history, licensing understandings, instructions for downloading, ...) about TGIF is available from the URL \url{bourbon.usc.edu/tgif/}.

Winedt
Winedt is an inexpensive ($40.00 per student user, $60 per educational user) editor for Intel-based machines running one or another version of Microsoft Windows. It has particular features that make it especially suitable for creating source files for \TeX/LaTeX. Information about licensing and downloads can be found at the URL \url{www.winedt.com}. Winedt is also available from CTAN, the \TeX/LaTeX distribution network accessible from links at \url{www.tug.org}. 
Xemacs
This flexible, open source text editor is available from a variety of sources and is protected under the terms of the GNU Public License. Downloads and information about the program can be found at the URL 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.
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


$\text{IDLHEAD}$, 11

$\text{MATHEMHEAD}$, 157

$\text{NRHEAD}$, 11, 255–258, 325, 327, 328, 331, 333, 335, 399–401, 404, 406, 454, 456, 457, 460–463

$\text{PYTHONHEAD}$, 15

A

aborting execution
  in C, 198
  in FORTRAN, 191
  in MATHEMATIC, 112
  in PYTHON, 16, 203
accuracy
  in finding roots, 434–435
  in solving ODEs, 294–295, 305, 308
  with conservation laws, 295, 308
acroread, 488
action statements, 171, 173–174
Adams/Bashforth/Moulton (ABM) methods, 296–297
adaptive methods
  for integrals, 377
  for ODEs, 295–296, 325, 331
adding text to graphics, see placing text in graphics
algorithms
  definition, 171
  for $a^2 + b^2 = c^2$, 224
  for adding, 225
  for bowling, 223
  for controlling loops, 180
  for counting, 180, 181
  for exchange, 180
  for finding extremes, 181, 182
  for integration, 372–380
  for Laplace’s equation, 182
  for ODEs, 288–297
  for parking lot, 223
  for root finding, 432–435
  for searching, 182, 224
  for sorting, 223
  for summing, 181
animation
  in MATHEMATIC, 153
  in PYTHON, 78–84
arithmetic operators
  in C, 196
  in FORTRAN, 189
  in MATHEMATIC, 115
  in PYTHON, 21
array indices
  in C, 196
  in FORTRAN, 189
  in MATHEMATIC, 120
  in pseudocode, 173
  in PYTHON, 23–27
arrays
  for data storage, 184
  in C, 196
  in FORTRAN, 189
  in MATHEMATIC, 119
  in pseudocode, 173
  in PYTHON, 24–27
scalar, see scalar fields
  vector, see vector fields
ASCII character codes, 5
ASCII text files, 8
aspell, 515–516
assigning values to variables
  in C, 196
INDEX

in FORTRAN, 189
in MATHEMATICA, 116–117
in pseudocode, 174
in PYTHON, 18

B
backsubstitution, 464
Bessel functions, 412, 473
binary files, 8
binary number system, 4–5
Biot-Savart law, 365
bisection, see root finding
bit, 4–5
black body radiation, 164, 411
blocking statements, see also control structures
in C, 197
in MATHEMATICA, 143
in pseudocode, 176, 186
in PYTHON, 35
Boolean values, see logical expressions
Boolean variables, see logical expressions
boundary value problems, 261, 272, 282–285, 477, 478
byte, 5–6

C
C, 1
C commands
fclose, 207, 216, 330, 333
fgets, 221
fopen, 198, 207, 216, 221, 330, 333, 335
fprintf, 198, 207, 216, 330, 333, 335
fscanf, 221
main, 195, 206, 215
printf, 195, 197, 199, 330, 402, 405, 406, 458–460
return, 201
scanf, 190, 330, 402, 458–460
C control structures
do/while, 200, 335
for, 195, 196, 198, 206, 207, 215, 216, 221, 330, 333, 402, 405
if/else, 198, 459
switch/case/default, 199
while, 200, 458, 459
C data types
char, 196, 221
file, 198, 206, 215, 221
float, 195, 196, 206, 215, 221
int, 195, 196, 206, 215, 221
C directives, 196
#define, 195, 196, 206, 215
#include, 195, 196, 206, 215
C functions
cos, 215, 462
exp, 405, 406
pow, 196, 206, 405, 406, 458, 459
sin, 206, 215, 460, 462
sqrt, 405, 406
user-defined, 200–201
C header files
math.h, 196, 206, 215, 402, 405, 406, 458, 460
nr.h, 258, 259, 333, 335, 404–406, 460–462
nrutil.h, 333, 335, 460–462
stdio.h, 196, 206, 215, 402, 405, 406, 458, 460
C programs
bisect.c, 458–459, 481
decay.c, 330, 347–348
decayrk4.c, 333, 348
decayrkqs.c, 335, 349
errqtrap.c, 405–407, 414, 415
ertrapzd.c, 404–405, 414
guessroot.c, 458
irrad.c, 204–206, 231–232
laplace.c, 195, 197
laplace_file.c, 198
pdens.c, 209–210, 236–237
quadpole.c, 218–219, 247–249
read_irrad.c, 252
sinrthbs.c, 460, 461
sincntnewt.c, 462–463
trapezoidal.c, 403, 417
wavegd.c, 213–214, 242–243
C special symbols\\n (new line), 197, 198, 207, 216, 330, 333, 335, 402, 458–460
* (multiplication), 196
+ (addition), 196
++ (incrementing index), 221, 402, 405
– (subtraction), 196
/ (division), 196
//.../** (comment), 196
%d (format), 197, 199, 207, 216, 221, 330, 405
%e (format), 197, 207, 216
%f (format), 197–199, 330, 333, 335, 405, 406, 458–460
%g (format), 221
%p (format), 405
&q (pointer), 199, 330, 335, 458–460, 463
braces, 197
cardioid, 52, 409
carrying capacity, 266, 277–278
Cartesian coordinates
in MATHEMATICA, 169
case sensitivity
in C, 196
in FORTRAN, 187
in MATHEMATICA, 112
in pseudocode, 172
in PYTHON, 16–18
cc, see UNIX commands
cells
in a MATHEMATICA notebook, 121
center of mass
  definition of, 352
  of cardioid, 409
  of semicircular plate, 353, 361
central processing unit, 3
chaos
  Lorenz system, 342
character codes, 4–5
character strings
  in MATHEMATICA, 114
  in PYTHON, 18
characteristic equation, 422
charging capacitor, 99
classification
  of integrals, 351
  of ODEs, 261
closing notebook
  in MATHEMATICA, 113
color
  in PYTHON, see also PYTHON keywords:
    color, edgecolor, colors, ..., 51, 52, 88–89
combining statements on one line
  in C, 196
  in MATHEMATICA, 112
  in pseudocode, 176
  in PYTHON, 16, 34, 54
command files
  in MATHEMATICA, see MATHEMATICA command files
  in PYTHON, see PYTHON scripts
command line editing
  in MATHEMATICA, 121
  in PYTHON, 16
commands
  in C, see C commands
  in FORTRAN, see FORTRAN commands
  in MATHEMATICA, see MATHEMATICA commands
  in pseudocode, see pseudocmds
  in PYTHON, see PYTHON commands
comments, see also documentation
  in C, 196
  in FORTRAN, 187
  in LaTeX, 516
  in MATHEMATICA, 145
  in pseudocode, 176–177, 180
  in PYTHON, 16, 41, 47
common storage
  in C, see global variables
  in FORTRAN, 194, 325
  in PYTHON, see global variables
compiler directives, 173
  in C, see C directives
  in FORTRAN, see FORTRAN directives
  in pseudocode, see pseudocode directives
compiling source file, see Local Guide, 172
  in C, 198, 206, 330, 333, 335, 403, 404, 407, 457, 459, 461, 462
  in FORTRAN, 191, 204, 323, 327, 328, 398, 400, 401, 451, 453, 454, 456
complex numbers
  in MATHEMATICA, 125–127
  in PYTHON, 28–29
Compton scattering, 100
conditions, see logical conditions
  constants, in MATHEMATICA, see MATHEMATICA constants
  in PYTHON, see PYTHON constants
contact information, 533–536
continuing statement on second line
  in C, 196
  in FORTRAN, 190
  in MATHEMATICA, 112
  in pseudocode, 176
  in PYTHON, 16
control characters, 4
control structures
  in C, see C control structures
  in FORTRAN, see FORTRAN control structures
  in MATHEMATICA, see MATHEMATICA control structures
  in pseudocode, see pseudocode control structures
  in PYTHON, see PYTHON control structures
convergence criteria, 184, 294–295, 375–377, 434–435
convert .eps/.ps to .pdf, 510–511
convert .pdf to .eps/.ps, 511
Cornu spiral, 356, 388–390
coupled oscillators, 268–269, 281–282, 336, 337
CPU, see central processing unit
Cramer’s rule, 464
creating arrays
  in PYTHON, 24–27
creating lists
  in PYTHON, 23
creating matrices
  in PYTHON, 27–28
creating sets
  in PYTHON, 23
creating tuples
  in PYTHON, 23
cross product
  in MATHEMATICA, 145–146, 170
  in PYTHON, 26, 97
current directory, see also default directory
  in PYTHON, 39, 40, 44, 45, 54, 58, 80, 82, 90–91
curve fitting, see least squares fitting
  to parabola, 168, 475
customizing
  matplotlib, 93–94
  MATHEMATICA, 155
  PYTHON, 86–88, 92–93
INDEX

cylindrical coordinates
  in MATHEMATICA, 169

D data analysis, 355
data files
  decay_c.dat, 331, 347–348
decay_f.dat, 324, 344
decayrk4_c.dat, 333, 348
decayrk4_f.dat, 326, 345
decayrkqs_c.dat, 335, 349
decayrkqs_f.dat, 327, 346
freefall.dat, 476
irrad_c.dat, 204–206, 221, 231–232, 252
irrad_f.dat, 204, 220, 230, 251
irrad_python.dat, 208, 222, 233, 252–253
laplace_c.dat, 198, 203, 227
laplace_f.dat, 191, 227
laplace_py.dat, 202
ldrum.dat, 81, 108–109
pdens_c.dat, 210, 236–237
pdens_f.dat, 209, 234–235
pdens_python.dat, 210, 238–239
quadpole_c.dat, 219, 247–249
quadpole_f.dat, 218, 245–246
quadpole_python.dat, 219, 249–250
radio.dat, 40, 106
rcdata.dat, 54, 106–107
rchardt.dat, 478
wavegd_c.dat, 213–214, 242–243
wavegd_f.dat, 213, 240–241
wavegd_python.dat, 216, 244
data types
  floating point, 6–7
  integer, 6–7
  string, 6–7
data typing
  explicit, 173
  implicit, 173
  in C, 196
  in FORTRAN, 189
  in pseudocode, 172–173, 175
  in PYTHON, 17
default directory, see also current directory, 9
  in MATHEMATICA, 144, 151, 152, 156–157
  in PYTHON, 39, 54, 84, 302, 311
defining ODEs
  for MATHEMATICA, 135, 275, 277, 278, 281, 282, 321
    in PYTHON, 301–302
deleting functions
  in MATHEMATICA, 118
deleting variables
  in MATHEMATICA, 117, 118, 129
  in PYTHON, 24, 34
demos
  in MATHEMATICA, 111, 113
diffraction patterns, see also interference patterns, 293, 413, 472, 474
  of square aperture, 58–67
dimensionless casting
  for plotting, 100
  of integrals, 353–355, 357, 365, 408, 411
  of ODEs, 138–139, 262–267, 269–271, 277–278, 421, 423
discretizing
  Laplace’s equation, 183
displaying data on screen
  in C, 195, 197
  in FORTRAN, 188, 190
  in MATHEMATICA, 112
  in pseudocode, 174
  in PYTHON, 16, 20–24
dividing statement between lines, see continuing statement on second line
documentation, see also comments
  in MATHEMATICA notebooks, 154–155
  in PYTHON py-files, 47
dot product
  in MATHEMATICA, 145–146, 170
  in PYTHON, 26, 97
double factorial, 12
drawing space curves
  with MATHEMATICA, 158–159
  with PYTHON, 85
drumhead, 80–82
dvipdfm, 488
dvips, 487, 505

E

electric field, 356
  in wave guide, 73–76, 211–216
  of dipole, 229
  of quadrupole, 217–219
  of relativistic particle, 100
electrostatic potential, 356
  of charged disks, 99, 408
  of charged rings, 356–357, 391–392, 395–396, 412
  of finite line charge, 362–364
  of spherically symmetric distribution, 409
elliptic integrals, 168, 355, 412
em dash, 496
em space, 493
en dash, 496
en space, 493
energy levels
  in quantum oscillator, see harmonic oscillator
  in quantum well, 285, 425, 442–444, 448–450, 472
equilibrium points, 420, 428–429
equivalent resistance, 475
error function, 356, 358, 380, 387–388, 395, 398, 399, 403, 404, 411
error messages
  in \LaTeX, 513–514
Euler’s method, 288–290, 297–299, 323–324, 329–331
  improved, see improved Euler method
evaluating integrals
  with C, 402–407
  with FORTRAN, 396–402
  with MATHEMATICA, 132, 359–371, 393–396
  with PYTHON, 380–392
exchange algorithm, 180
executing program, see Local Guide
  exiting
  from MATHEMATICA, 113
  from PYTHON, 16
expressing infinity
  in MATHEMATICA, 360
expressions
  in MATHEMATICA, 114–116

F

f77, see UNIX commands
Fabry-Perot interferometer, 99
factorial, 12, 21, 122
Fibonacci numbers, 97
field mapping, 102, 103
finding eigenvalues/eigenvectors
  with MATHEMATICA, 130–131, 165, 166
  with PYTHON, 36–38, 97, 98
finding roots, see root finding
finite difference approximations for derivatives, 225
fitting data, see curve fitting, least squares fitting
floating point number
  double precision, 7
  single precision, 6
fonts, see also Greek letters
formatting data files, 184–185
formatting output
  in C, 197, 207, 216
  in FORTRAN, 190
  in pseudocode, 174
  in PYTHON, 43, 201, 298
FORTRAN, 1
  position of statements on line, 189
FORTRAN commands
  close, 205, 214, 220, 324, 326
  format, 192, 193, 205, 214, 326, 327
  open, 191, 205, 214, 220, 323, 326, 327
  read, 192, 193, 220, 323, 397, 451, 452, 454
  write, 188, 190–192, 205, 214, 323, 324, 326–328, 397, 399, 451–454
FORTRAN control structures
  call, 193, 326–328, 399, 457
  do while/endo, 193, 328, 451, 453
  do/endo, 188–191, 205, 213, 214, 220, 324, 326, 397, 399
  do/endo, 204
  function/return, 194, 397, 399, 401
  if/then/else/endif, 191, 453
  program/endif, 189, 204, 212–213, 399, 401, 454
  subroutine/return, 193, 325, 456
FORTRAN data types
  character, 189, 220
  double precision, 189
  integer, 189
  real, 189
FORTRAN directives
  common, 194, 325, 326, 328
  dimension, 188, 204, 213, 220, 325, 326, 328
  external, 399, 401, 454
  implicit, 189
  parameter, 188, 189, 204, 213
FORTRAN functions
  cos, 213, 456
  exp, 397, 399, 401
  sin, 204, 213, 454, 456
  sqrt, 397, 399, 401
  user-defined, 194, 454
FORTRAN programs
  bisect.f, 452–453, 480
  decay.f, 323, 344
  decayrk4.f, 326, 345
  decayrkqs.f, 328, 346
  errqtrap.f, 400–401, 414
  errtrapzd.f, 399–400, 414
  guessroot.f, 451
  irrad.f, 203–204, 230
  laplace.f, 188, 191
  laplace_file.f, 191
  pdens.f, 209, 234–235
  quadpole.f, 218, 245–246
  read_irrad.f, 251
  sinrtbis.f, 454
  sinrtnewt.f, 456
  trapezoidal.f, 398, 416
  wavegd.f, 212–213, 240–241
FORTRAN special symbols
  ! (comments), 187
  * (multiplication), 189
  ** (exponentiation), 189, 204, 397, 399, 401, 451, 452
  + (addition), 189
  - (subtraction), 189
  / (division), 189
  A (in format statements), 192, 323, 397, 401, 451, 452
  C (comments), 187
  E (in format statements), 190
  F (in format statements), 190, 191, 323, 326, 327, 397, 399, 401, 454
  I (in format statements), 190, 399
FORTRAN subroutines, 193–194
  user-defined, 455
four-slit interference pattern, 100
Fourier series, 359, 369–371, 409
Fraunhofer diffraction, 203–208
Fresnel integrals, see also Cornu spiral, 356, 413
functions
  in C, see C functions
  in FORTRAN, see FORTRAN functions
  in MATHEMATICA, see MATHEMATICA functions
  in PYTHON, see PYTHON functions,
    PYTHON scripts
typical available, 174

G
gamma function, 12–13
Gauss-Kronrod rule, see quadrature
Gaussian distribution, 355, 411
Gaussian elimination, 464–465
Gaussian integration, see quadrature
gedit, 1
ghostview, 488
global positioning, 475
global truncation error, see truncation error,
global
global variables
  in C, 201, 332
  in FORTRAN, see common storage
  in PYTHON, 45–47, 390
golden ratio, 471
gravitational force, 270
Greek letters, see also fonts
  in \LaTeX, 496–497
  in MATHEMATICA, 114
  in PYTHON, 61–62, 84

H
hard copy
  from MATHEMATICA, 149–152
  from PYTHON, 57–58
Helmholtz coil, 364–368
help, see on-line help
hexadecimal number system, 5
high-pass filter, 54–57
high-resolution graphing
  in MATHEMATICA, 146–149
  in PYTHON, 30–33, 48–78
Hooke’s law, 421
hyperlinks, 509–510

I
IDL, 1
IDL contact information, 533
improved Euler method, 290–291, 298
indices in \LaTeX, 507
indicial equation, 285
inert commands
  in MATHEMATICA, see suppressing evaluation in MATHEMATICA
inhomogenieties in linear equations, 464
initial value problems, 261
initialization files
  in MATHEMATICA, 155
  in PYTHON, 92–93
integer
  signed, 6
  unsigned, 6
interactive plotting in PYTHON, 90
interference patterns, see also diffraction patterns
  Fabry-Perot, 99
  from four slits, 100
internal storage of expressions
  in MATHEMATICA, 115–116
interpolation points, 373, 374
interpreting source code, see also Local Guide, 172
interrupt
  in MATHEMATICA, 112
  in PYTHON, 16
ispell, 515–516

K
keywords
  in PYTHON, see PYTHON keywords
Kirchoff’s laws, 475
Knuth, Donald, 483
Kronecker delta, 358

L
Lagrange equations
  for 2D harmonic oscillator, 166
  for compound pendulum, 167
Laguerre polynomials, 164
Laguerre’s method, see root finding
Lamport, Leslie, 483
Laplace transforms, 13–14, 273–274, 338
definition, 13, 134, 273
for ODEs, 273
in MATHEMATICA, 134, 168
Laplace’s equation, 478
  in C, 194–198, 227
  in FORTRAN, 187–191, 227
  in pseudocode, 182–184
  in PYTHON, 201–203
\LaTeX, 1
  accents in, 519
  counters in, 520
  dashes in, 495
  math mode in, 496–497
  quotation marks in, 496
  special characters in, 520
latex, 486, 487, 505
\LaTeX commands
\verb|(italic correction), 492\n|\verb|(new line), 494, 499\n|\verb|(negative thin space), 497\n|\verb|(thin space), 497\n|\verb|(allow hyphen) , 496\n|\verb|(medium space), 497\n|\verb|(thick space), 497\n|\verb|\addcontentsline, 506–507\n|\verb|\addtocontents, 506–507\n|\verb|\Alph, 498, 521\n|\verb|\alph, 498, 521\n|\verb|\and, 512\n|\verb|\arabic, 498\n|\verb|\author, 516\n|\verb|\bf, 516\n|\verb|\bfseries, 492\n|\verb|\boldmath, 497\n|\verb|\cal, 519\n|\verb|\caption, 499, 500, 502, 505\n|\verb|\centerline, 502\n|\verb|\chapter, 496\n|\verb|\chaptername, 523\n|\verb|\char, 520\n|\verb|\clearpage, 493\n|\verb|\contentsname, 523\n|\verb|\date, 516\n|\verb|\definecolor, 509, 513\n|\verb|\documentclass, 484, 486, 489, 510, 516\n|\verb|\documentstyle, 484\n|\verb|\fbox, 520\n|\verb|\figurename, 522\n|\verb|\footnote, 496, 516\n|\verb|\footnotetext, 493\n|\verb|\frac, 516\n|\verb|\framebox, 520\n|\verb|\hbox, 493, 494\n|\verb|\hbox*, 493\n|\verb|\Huge, 493\n|\verb|\huge, 493\n|\verb|\IfFileExists, 513\n|\verb|\ifthenelse, 512–513\n|\verb|\includegraphics, 500–503, 510\n|\verb|\index, 507\n|\verb|\indexname, 523\n|\verb|\input, 519\n|\verb|\int, 516\n|\verb|\item, 497–498, 521\n|\verb|\itshape, 492\n|\verb|\label, 496, 499, 500, 502, 505, 516\n|\verb|\labelitem, 498\n|\verb|\labelitemi, 498\n|\verb|\labelitemii, 498\n|\verb|\labelitemiii, 498\n|\verb|\labelitemiv, 498\n|\verb|\LARGE, 493\n|\verb|\Large, 493, 516\n|\verb|\large, 493\n|\verb|\LaTeX, 516\n|\verb|\listoffigures, 506–507\n|\verb|\listoftables, 506–507\n|\verb|\makebox, 520\n|\verb|\makeindex, 507–508\n|\verb|\maketitle, 516\n|\verb|\mathbf, 497\n|\verb|\mathcal, 519\n|\verb|\mathit, 497\n|\verb|\mathrm, 497\n|\verb|\mathsf, 497\n|\verb|\mailbox, 496, 497, 520\n|\verb|\mdseries, 492\n|\verb|\needspace, 523\n|\verb|\needspace*, 523\n|\verb|\newboolean, 512–513\n|\verb|\newcommand, 517\n|\verb|\newcounter, 521\n|\verb|\newlength, 494\n|\verb|\newpage, 493\n|\verb|\noindent, 493, 516\n|\verb|\normalfont, 492\n|\verb|\normalsize, 493\n|\verb|\not, 512\n|\verb|\or, 512\n|\verb|\pageref, 496, 499, 500\n|\verb|\pagetitle, 516, 519\n|\verb|\parbox, 520\n|\verb|\part, 496\n|\verb|\partial, 516\n|\verb|\partial, 516\n|\verb|\printindex, 507\n|\verb|\quad, 493\n|\verb|\quad, 493\n|\verb|\ref, 496, 499, 500, 516\n|\verb|\renewcommand, 495, 498, 518, 521, 522\n|\verb|\rmfamily, 492\n|\verb|\Roman, 498, 522\n|\verb|\roman, 498, 522\n|\verb|\rule, 520\n|\verb|\scriptsize, 493\n|\verb|\scshape, 492\n|\verb|\section, 496\n|\verb|\settowidth, 494\n|\verb|\settoheight, 493\n|\verb|\setboolean, 512–513\n|\verb|\setcounter, 521\n|\verb|\setlength, 489, 493, 516\n|\verb|\settodepth, 493\n|\verb|\setlength, 493\n|\verb|\sffamily, 492\n|\verb|\slshape, 492\n|\verb|\small, 493\n|\verb|\subsection, 496\n|\verb|\subsubsection, 496\n|\verb|\tablename, 523\n|\verb|\tableofcontents, 506–507\n|\verb|\textbf, 492\n|\verb|\textit, 492\n|\verb|\textmd, 492\n|\verb|\textnormal, 492\n|\verb|\textsc, 492\n|\verb|\textsf, 492
\textsl, 492
\texttt, 492
\textup, 492
the..., 521–522
theenumi, 498
thenumii, 498
theenumii, 498
theenumiv, 498
\thispagestyle, 516, 519
tiny, 493
title, 516
today, 517, 518
\ttfamily, 492
twocolumn, 522
typein, 513
typout, 513
upshape, 492
\usepackage, 495, 501, 507
value, 521
verb, 492, 516, 520
\verbatiminput, 520
\vspace, 493, 500
\vspace*, 493
\LaTeX contact information, 483, 535
\LaTeX counters
enumi, 498, 521
enumii, 498
enumiii, 498
figure, 521
footnote, 521
page, 521
secnumdepth, 506
section, 521
subsection, 521
subsubsection, 521
table, 521
tocdepth, 506
\LaTeX declarations, see \LaTeX commands
\LaTeX document classes, 485
article, 489, 496
book, 489, 496
report, 489
slides, 489
\LaTeX environments, 494–495
\(...\) (in-line equation), 496
\{...\} (displayed equation), 496
$...$ (in-line equation), 496
array, 523
center, 494, 499, 501, 505, 516
comment, 520
displaymath, 496
document, 485, 486, 516
enumerate, 497–498, 520, 521
enarray, 496
equation, 496, 516
figure, 500–503, 505
flushleft, 516
itemize, 497–498
minipage, 520
pict2e, 506
picture, 500, 506
quotation, 494
quote, 494
table, 498–500
tabular, 498–500
tikzpicture, 503–506
\LaTeX length parameters, 490
\baselineskip, 493, 495
\baselinestretch, 495
\bottomfraction, 521
\evensidemargin, 489, 516
\textfraction, 521
\itemsep, 498
\marginparsep, 508
\parindent, 489, 516
\parskip, 498
\parskip, 489, 498, 516
\textfraction, 521
\textwidth, 489, 508, 516
\topfraction, 521
\topmargin, 489, 516
\LaTeX Manual, 483
\LaTeX output from MATHEMATICA, 152–153
\LaTeX packages, 495
alltt, 519
color, 509, 513
graphicx, 500–503, 509–510
hyperref, 509–510, 513
ifthen, 512–513
imakeidx, 507–508
makeidx, 507–508
needspace, 523
notebook2e, 153
showidx, 508
tikz, 500, 503–506
verbatim, 519–520
\LaTeX page previewers, see also xdvi, yap, 514–515
\LaTeX preamble, 489–491
\LaTeX sample, 485–488, 516–517
\LaTeX special symbols
'"' (closing double quotation), 496
--- (em dash), 495
-- (en dash), 495
- (dash), 495
- (minus sign), 496
" (hard space), 495
'"' (opening double quotation), 496
\LaTeX templates, 490
\LaTeX type size, 492
\LaTeX type styles, 491–492
lawn mower, 470
least squares fitting, 476, 478
INDEX

Legendre polynomials, 162, 380, 410, 472
Lennard-Jones potential, 474
licensing of software, 14
line labels
  in MATHEMATICA, 112
linking object code, see Local Guide, 172
lists
  in MATHEMATICA, 114–116, 119–120, 281
  in PYTHON, 22–24
Local Guide, 2, 8, 9, 11, 14–16, 40, 58, 111, 112,
  145, 155, 157, 187, 191, 194, 197, 198,
  201, 203, 204, 206, 208, 213, 214, 216,
  218, 219, 255, 257, 258, 297, 319, 322,
  329, 362, 380, 396, 398, 402, 403, 405,
  435, 450, 451, 457, 483, 485–489, 491,
  501, 514, 516
logical conditions, see logical expressions
logical expressions
  in C, 196
  in FORTRAN, 189
  in \LaTeX, 512–513
  in MATHEMATICA, 143–144
  in pseudocode, 174–175
  in PYTHON, 35–36
logical operators, see logical expressions
logistic growth, 266, 277–278
long integer
  signed, 6
  unsigned, 6
loops
  in C, 196
  in FORTRAN, 189
  in MATHEMATICA, 123, 128, 142–143
  in pseudocode, 177–178
  in PYTHON, 33–36
Lorentz distribution, 98, 408, 412
Lorentz lineshape, 45
lp, see UNIX commands
LSODE contact information, 533
LU decomposition, 96, 465–466

M

machine code, 172
Macintosh OS, 1
Mac\TeX
  contact information, 533
magnetic field, 356
  in wave guide, 73–76, 103, 211–216
  of circular loops, 49, 101, 228, 364–368, 413
  of Helmholtz coil, 364–368
  of solenoid, 101
magnetic vector potential, 356
MAPLE, 1
MAPLE contact information, 534
MATHEMATICA, 1
mathematica, 112
MATHEMATICA command files, 144–146
  crossdot, 145, 170, 362, 365
  testmultiply, 145
MATHEMATICA commands, see also
  MATHEMATICA functions, 114
  Animate, 153
  Apart, 124, 160, 168
  Append, 157
  Black, 147, 148
  Boole, 143
  Cancel, 138
  Clear, 117, 118
  CoefficientArrays, 283
  CoefficientList, 162, 286
  Collect, 137, 162, 276, 282, 287
  ComplexExpand, 125–127, 137
  Conjugate, 125
  ContourPlot, 149
  Cross, 146
  Curl, 140, 169, 170
  D, 132–136, 146, 162, 165, 321, 429
  Dashing, 147, 148
  Delete, 157
  Det, 130, 284
  DiagonalMatrix, 120
  Directory, 156
  Display, 151
  Div, 140, 169, 170
  Dot, 146
  Dsolve, 135–139, 274–287
  Eigensystem, 131
  Eigenvalues, 131
  Eigenvectors, 131
  Element, 284, 369–370
  Evaluate, 321
  Exit, 122
  Expand, 123–125, 128, 137, 139, 160, 282,
    431, 446
  Export, 151, 152
  ExpToTrig, 126, 127, 282
  Factor, 123, 137, 160, 432
  Remove, 129
  FileNames, 157
  FilePrint, 146
  FindRoot, 367, 445–449, 466, 469
  FullForm, 123, 124, 132, 133, 161
  Get, 144, 145, 362, 365
  Grad, 140, 169, 170
  Graphics, 158, 450
  Head, 123, 133
  HoldForm, 133
  IdentityMatrix, 120
  Insert, 157
  Integrate, 132, 134, 359–371, 393, 394
  Inverse, 130
  InverseLaplaceTransform, 134, 168
  LaplaceTransform, 134
  Laplacian, 140, 169, 170
  Limit, 134, 277, 280, 360
  Line, 450
  LinearSolve, 129, 466
ListPlot, 395, 396, 447, 450
lucross, 145–146, 365
ludot, 145–146, 362, 365
Map, 137, 449
MatrixForm, 120, 130, 166, 284
N, 122, 129, 361, 367, 368, 393, 394, 428, 429, 444, 466
Names, 117
NDSolve, 320–322
NIntegrate, 393–396
Normal, 133, 134, 283, 363, 364, 367, 369, 431, 432
NSolve, 129, 444–450, 466
Numerator, 367
Off, 156
On, 157
Options, 114, 149, 158
ParametricPlot, 148
ParametricPlot3D, 159
Part, 123–125, 161
Plot, 146–148, 151, 153, 158, 278, 280, 321, 366, 371, 430
Plot3D, 149, 153
Prepend, 152
Print, 123, 128, 144–145
Protect, 118
Put, 152
PutAppend, 152
Quit, 113, 121, 122, 125, 127–129, 131, 133, 134, 138, 149, 276, 361, 429, 430
Range, 119, 395
ReleaseHold, 134
Remove, 117, 118
Replace, see /., 124
ReplacePart, 125, 157, 161
Rest, 449
Root, 427
Series, 132, 134, 276, 361, 363, 364, 366, 368, 369, 431, 432
SetDirectory, 156–157
SetOptions, 157
Show, 158, 450
Solve, 128, 129, 137, 275, 284, 286, 287, 360, 367, 427–431, 444, 466, 469
StyleForm, 147, 148, 278, 280
Sum, 371
Table, 119, 120, 286, 287, 322, 395, 447, 449, 450
TeXForm, 152
Text, 147, 158
Thickness, 147, 148, 278, 280, 366, 395, 396, 430, 447, 450
Tr, 130
Transpose, 130
TrigExpand, 127, 128
TrigReduce, 127, 128
TrigToExp, 127
Union, 281
Unprotect, 118
ZeroMatrix, 166
MATHEMATICA constants, 114
E (2.71828...), 114
False, 114
I (\(\sqrt{-1}\)), 114, 125, 126, 281
Infinity (\(\infty\)), 114, 132, 360
Pi (\(\pi\)), 114, 118, 119, 122, 149, 284, 394, 395, 447
True, 114, 276, 277
MATHEMATICA contact information, 534
MATHEMATICA control structures
For, 123, 128, 142–143, 395, 447, 449, 450
If, 132, 143–144
While, 143
MATHEMATICA domains
Integers, 284, 369
MATHEMATICA functions, see also MATHEMATICA commands, 114–115, 117–118
Abs, 126
ArcTan, 126
Arg, 126
ConditionalExpression, 132, 445
Cosh, 125
cot, 449
Exp, 126, 132, 134, 146, 277, 394, 395, 430, 447
Factorial, 122
Im, 125, 126, 132
LegendreP, 163
Log, 364
Re, 125, 126
Sin, 118–120, 125, 127, 133, 134, 146, 149, 151, 153, 158, 159, 282, 359, 362, 365, 369–371, 430, 449
Sinh, 125
Sqrt, 122, 130, 137, 281, 282, 360, 362, 364, 368, 394, 395, 429, 449
MATHEMATICA heads
Plus, 115, 116, 123, 124, 133
Power, 116, 123, 124, 133
Rational, 132, 133
SeriesData, 132
Times, 115, 116, 123, 124, 133
MATHEMATICA logical expressions, see logical expressions in MATHEMATICA
MATHEMATICA manuals, 111
MATHEMATICA notebooks, 154–155
MATHEMATICA options, 114
AccuracyGoal, 322, 393, 394, 446
AnimationRate, 153
AspectRatio, 148, 149
Assumptions, 114, 132, 361, 363, 364

546
INDEX
AxesLabel, 147, 148, 151, 278, 280, 321, 366, 371, 430, 447, 450
AxesOrigin, 450
Cartesian, 139–141
ColorFunction, 149
Contours, 149
ContourShading, 149
cylindrical, 139–141
DisplayFunction, 159
Epilog, 147
FontFamily, 147, 148, 278, 280
FontSize, 147, 148, 278, 280
FontWeight, 147, 148, 278, 280
FormatType, 151
GaussPoints, 393
GridLines, 395, 396, 450
MaxIterations, 446
MaxRecursion, 393
MaxSteps, 322
MaxStepSize, 322
Method, 393, 394
PlotJoined, 395, 396, 447, 450
PlotLabel, 147, 148, 278, 280, 395, 396
PlotPoints, 147–149
PlotRange, 114, 147, 148, 153, 278, 280, 366, 371, 396, 430, 450
PlotStyle, 147, 148, 159, 278, 280, 321, 366, 371, 395, 396, 430, 447, 450
PrecisionGoal, 322, 393, 394
Spherical, 139–141
TargetFunctions, 114, 126–127
TextStyle, 114
Thickness, 159
WorkingPrecision, 322, 393, 394, 446
MATHEMATICA packages, 141–142
LinearAlgebra
MatrixManipulation, 166
MATHEMATICA special symbols
! (factorial), 122
' (differentiation), 135, 138, 275–278, 281, 282, 286
(* ... *) (comments), 145
* (multiplication), 114, 115, 117, 122, 130
+ (addition), 114, 115, 121
- (subtraction), 114, 115, 122
. (clear variable), 118
. (matrix multiplication), 130
/ (division), 114, 115, 121
// (convert form), 120, 122, 367, 428
:= (function definition), 114, 117–118, 133, 370, 449
= (assignment), 117, 119
? (help), 113, 117, 118
?? (more help), 113
[...] (element of list), 119, 120, 128, 136, 137, 275, 277, 280, 284, 286, 287, 360, 395, 430, 431, 449, 450
#1 (generic variable), 427
% (previous output), 120, 123, 125, 127–130, 132, 134, 287, 363, 367, 369, 429, 431, 447
%% (second previous output), 123, 132
%%% (third previous output), 123
%n (output on line n), 116, 123, 287
_ (pattern), 117–118, 133, 370, 395
>> (direct output to file), 152
>>> (append output to file), 152
^ (exponentiation), 115, 117, 118, 122, 123
Out[n] (output on line n), 116, 123
MATHEMATICA statements
termination of, 112
MATHEMATICA system variables, 114
$BaseDirectory, 155
$HomeDirectory, 114
$IgnoreEOF, 114
$MachinePrecision, 322, 394
$messageGroups, 156
$Packages, 114, 141
$Path, 114, 157
$RecursionLimit, 114
$TextStyle, 114
$TopDirectory, 114
$UserBaseDirectory, 155
mathematical functions
in C, see C functions
in FORTRAN, see FORTRAN functions
in MATHEMATICA, see MATHEMATICA functions
in PYTHON, see PYTHON functions
MATLAB, 1
MATLAB contact information, 534
matrices
in MATHEMATICA, 119, 120, 130–131
in PYTHON, 27–28
matrix multiplication
in MATHEMATICA, 130
in PYTHON, 27–28
max-min problems, 420
MAXIMA, 1
MAXIMA contact information, 533
Maxwell-Boltzmann distribution, 411
method of bisection, see root finding
method of Frobenius, 285–287
method of least squares, see least squares fitting
midpoint method, 293
midpoint rule, see quadrature
MiKTeX, 1
MiKTeX contact information, 534
ML (left mouse button), 9
MM (middle mouse button), 9
moment of inertia
definition of, 353
of cardioid, 409
of semicircular plate, 353, 362, 386–387
motion
of membrane, 104
of string, 104
MR (right mouse button), 9
MUDPACK contact information, 534
Müller’s method, see root finding
multi-way selection
in C, 198–199
in FORTRAN, 191–192
in pseudocode, 177
multistep methods
for ODEs, 296–297
N
NAG library contact information, 534
named common
in C, see global variables
in FORTRAN, see common storage
in PYTHON, see global variables
natural frequencies
of spherical cavity, 472
of stiff bar, 472
of three-mass system, 471
of two-mass system, 421–422, 429–430, 438–439
Newton’s method, see root finding
Newton’s second law, 351, 352, 421, 423
normal distribution, see Gaussian distribution
normal modes
of membrane, 58–67
of string, 104
numerical integration, see quadrature
Numerical Recipes, 255–260
for ODEs
algorithm routines, 324, 331
driver routines, 325, 331
stepper routines, 325, 331
in C, 331–336, 460–463
rtbis.c, 460–463
rtnewt.c, 460–463
rtsafe.c, 460
zbrak.c, 463
broydn.c, 469
bsstep.c, 331
caldat.c, 258
flmoon.c, 258–259
gaussj.c, 467
julday.c, 258
lubksb.c, 467
ludcmp.c, 467
mmid.c, 331
mnewt.c, 469
newt.c, 469
nrutil.c, 258, 333, 335, 406, 461, 462
qromb.c, 407, 414
qsimp.c, 407, 414
qtrap.c, 405–407
rk4.c, 331–333
rkck.c, 331
rkqs.c, 331, 334–336
rtsafe.c, 463
svbksb.c, 467
svdcmp.c, 467
trapzd.c, 404–407
tridag, 467
in FORTRAN, 324–329, 453–457
rtbis.f, 453–456
rtnewt.f, 453, 455–456
rtsafe.f, 453
zbrak.f, 456–457
broydn.f, 469
bsstep.f, 325
caldat.f, 257
flmoon.f, 256–257
gaussj.f, 467
julday.f, 257
lubksb.f, 467
ludcmp.f, 467
mmid.f, 324
mnewt.f, 469
newt.f, 469
nrutil.f, 258, 333, 335, 406, 461, 462
qromb.f, 401, 414
qsimp.f, 401, 414
qtrap.f, 400–401
rk4.f, 324–326
rkck.f, 324
rkqs.f, 325, 327–329
rtsafe.f, 456
svbksb.f, 467
svdcmp.f, 467
trapzd.f, 399–401
tridag.f, 467
NUMERICAL RECIPES contact information, 534
nybble, 5
O
object code, 172
object-image relations, 99
octal number system, 5
OCTAVE, 1
OCTAVE contact information, 534
on-line help
in MATHEMATICA, 113
in PYTHON, 48
INDEX

ordinary differential equations
with C, 329–336
with FORTRAN, 322–329
with MATHEMATICA, 135–139, 274–287, 320–322
with PYTHON, 297–320
orthogonal functions, 358–359
OzTeX contact information, 535

P

page previewer
in \LaTeX, 514–515
partial fractions, 124, 168, 337
passwords, see Local Guide
path (in directory tree), 8
path specification
in MATHEMATICA, 156
in PYTHON, 16
PDF files, 487–488
pdffcrop, 510–511
pdflatex, 487, 504, 509–510
pdftops, 511
pendulum, simple, see simple pendulum
phases of moon, 256–259
pivotal condensation, 465
pivoting, 465
placing text in graphics
in MATHEMATICA, see MATHEMATICA commands, Text; MATHEMATICA options, Epilog
in PYTHON, see PYTHON commands, matplotlib.pyplot.text
Planck radiation law, 100
polar coordinates
in MATHEMATICA, 126–127, 139–141
in PYTHON, 52–53, 67–68
population biology, 266, 277–278, 343
preallocation of arrays
in PYTHON, 34, 70
predictor-corrector methods, 291
procedure, 179
program, 171
projectile motion, see trajectories
properties
in PYTHON, see PYTHON keywords
proprietary software, 14
protected variables
in MATHEMATICA, 118–119
in PYTHON, see PYTHON special symbols, PYTHON constants
ps2pdf, 488, 505, 510–511
pseudocode comments, 176–177, 180
pseudocode control structures, 171, 176–179
CASE-END_CASE, 177
EXECUTE, 178
FOR-THRU-DO, 178, 184–186
FOR-THRU-DO-STEP, 178
IF-THEN-ELSE-END_IF, 3, 4, 176, 181, 182
LOOP-END_LOOP, 3, 4, 177, 180–182
REPEAT-UNTIL, 178
WHILE-DO, 178
pseudocode directives
DIMENSION, 173, 184, 186
pseudocommands
ATTACH, 186
CLOSE, 186
READ, 174
WRITE, 174, 184, 186
py-files, see PYTHON scripts, PYTHON functions
PYTHON, 1
python, 16, 39, 80, 82
PYTHON attributes, 30
add_subplot, 82, 85, 88
append, 34, 40, 81, 91
axis, 87
clear, 82
close, 40, 81, 202, 208, 217, 222
dtype, 25, 30
extend, 34
format, 43, 202, 298
grid, 79
imag, 28
insert, 34
itemsize, 25, 30
ndim, 25, 30
plot, 54, 79, 85, 87, 306
plot_surface, 59, 61–63, 65–66, 68, 70, 82, 88
plot_wireframe, 59–61
pos, 84
quiver, 77, 104
read, 39, 40, 44, 45, 47, 80, 82, 84, 302, 305, 306, 311, 314, 319, 385, 386, 389, 437–439
readline, 40–41, 222
readlines, 40
real, 28
reshape, 25, 30, 438
set_data, 79
set_title, 54, 85, 87, 306
set_xlabel, 54, 60, 62, 65, 70, 306
set_xlim, 82
set_xticks, 85, 88
set_ylabel, 54, 60, 62, 65, 70, 306
set_ylim, 82
set_yticks, 85, 88
shape, 25, 30
size, 25, 30, 392
sort, 96
split, 40, 95
subplots_adjust, 53, 54, 87, 306
velocity, 83, 84
view_init, 62, 67, 77
write, 42–43, 202, 208, 217
PYTHON class attributes, see PYTHON attributes, 30
PYTHON color codes, 52
PYTHON command files, see PYTHON scripts
PYTHON commands, see also PYTHON functions, PYTHON scripts

<table>
<thead>
<tr>
<th>animation</th>
<th>FuncAnimation, 78–82</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>17</td>
</tr>
<tr>
<td>break</td>
<td>36</td>
</tr>
<tr>
<td>clear</td>
<td>34</td>
</tr>
<tr>
<td>continue</td>
<td>36</td>
</tr>
<tr>
<td>del</td>
<td>24, 34</td>
</tr>
<tr>
<td>dir</td>
<td>24</td>
</tr>
<tr>
<td>exec</td>
<td>39, 44, 45, 47, 80, 82, 84, 302, 304–306, 311, 314, 385, 386, 389, 437–439</td>
</tr>
<tr>
<td>execfile</td>
<td>39, 44, 45, 47, 80, 82, 302, 304–306, 311, 314, 385, 386, 389, 437–439</td>
</tr>
<tr>
<td>exit</td>
<td>16</td>
</tr>
<tr>
<td>float</td>
<td>17, 81, 222</td>
</tr>
<tr>
<td>getattribute</td>
<td>18–19</td>
</tr>
<tr>
<td>global</td>
<td>46, 47, 79</td>
</tr>
<tr>
<td>globals</td>
<td>24</td>
</tr>
<tr>
<td>help</td>
<td>47</td>
</tr>
<tr>
<td>hex</td>
<td>17</td>
</tr>
<tr>
<td>import</td>
<td>19–21, 36, 44, 45, 48, 52, 54, 65</td>
</tr>
<tr>
<td>input</td>
<td>80, 94–95</td>
</tr>
<tr>
<td>int</td>
<td>17, 80, 95, 222</td>
</tr>
<tr>
<td>isinstance</td>
<td>95</td>
</tr>
<tr>
<td>join</td>
<td>91</td>
</tr>
<tr>
<td>len</td>
<td>22, 23</td>
</tr>
<tr>
<td>locals</td>
<td>24</td>
</tr>
<tr>
<td>matplotlib</td>
<td>is_interactive, 90</td>
</tr>
<tr>
<td>rcParams</td>
<td>93</td>
</tr>
<tr>
<td>matplotlib.pyplot</td>
<td>axvline, 89</td>
</tr>
<tr>
<td></td>
<td>axes, 60, 62, 65, 70, 77, 79, 85</td>
</tr>
<tr>
<td></td>
<td>axhline, 89</td>
</tr>
<tr>
<td></td>
<td>axis, 64, 70, 88, 89, 390</td>
</tr>
<tr>
<td></td>
<td>box, 88, 444</td>
</tr>
<tr>
<td></td>
<td>clabel, 64</td>
</tr>
<tr>
<td></td>
<td>contour, 59, 63–64, 68, 70, 76</td>
</tr>
<tr>
<td></td>
<td>contourf, 64</td>
</tr>
<tr>
<td></td>
<td>errorbar, 56</td>
</tr>
<tr>
<td>figure</td>
<td>60, 62, 65, 68, 70, 75, 77, 79, 82, 85, 88</td>
</tr>
<tr>
<td></td>
<td>get_backend, 90</td>
</tr>
<tr>
<td>grid</td>
<td>32, 33, 41, 50, 52, 57, 308, 315, 388, 390, 392, 438, 439, 441</td>
</tr>
<tr>
<td></td>
<td>ioff, 90</td>
</tr>
<tr>
<td></td>
<td>ion, 90</td>
</tr>
<tr>
<td>meshgrid</td>
<td>69</td>
</tr>
<tr>
<td>plot</td>
<td>32–33, 41, 45, 49–57, 88, 89, 97, 298, 303, 304, 308, 311, 314, 388, 389, 392, 438, 439, 441</td>
</tr>
<tr>
<td>polar</td>
<td>52</td>
</tr>
<tr>
<td>quiver</td>
<td>75–76</td>
</tr>
<tr>
<td>semilogx</td>
<td>54–57</td>
</tr>
<tr>
<td>show</td>
<td>32–33, 41, 45, 49, 50, 52, 60, 62, 65, 68, 70, 82, 85, 88, 89, 298, 303, 304, 388, 392, 438, 439, 441</td>
</tr>
<tr>
<td>subplots</td>
<td>53–54, 87, 306</td>
</tr>
<tr>
<td>text</td>
<td>41, 57, 89, 298, 315, 317</td>
</tr>
<tr>
<td>tick_params</td>
<td>303, 304, 311, 315, 317, 388, 390, 438, 439, 441, 444</td>
</tr>
<tr>
<td>title</td>
<td>32, 33, 50, 52, 60, 62, 89, 298, 303, 304, 308, 388, 390, 392</td>
</tr>
<tr>
<td>xlabel</td>
<td>32, 33, 50, 52, 75, 298, 303, 304, 308, 315, 317, 388, 390, 392, 438, 441</td>
</tr>
<tr>
<td>ylabel</td>
<td>32, 33, 50, 52, 75, 298, 303, 304, 308, 315, 317, 388, 390, 392, 438, 439, 441</td>
</tr>
<tr>
<td>yticks</td>
<td>88</td>
</tr>
<tr>
<td>zlabel</td>
<td>32, 33, 50, 52, 75, 298, 303, 304, 308, 315, 317, 388, 390, 392, 438, 439, 441</td>
</tr>
<tr>
<td>zticks</td>
<td>88</td>
</tr>
<tr>
<td>meshgrid</td>
<td>69</td>
</tr>
<tr>
<td>numpy</td>
<td>append, 97, 298, 389, 438, 441</td>
</tr>
<tr>
<td></td>
<td>argmax, 441</td>
</tr>
<tr>
<td></td>
<td>array, 25, 26, 40, 73</td>
</tr>
<tr>
<td></td>
<td>concatenate, 37, 81</td>
</tr>
<tr>
<td></td>
<td>cross, 26</td>
</tr>
<tr>
<td></td>
<td>dot, 26</td>
</tr>
<tr>
<td></td>
<td>eye, 26</td>
</tr>
<tr>
<td></td>
<td>loadtxt, 54</td>
</tr>
<tr>
<td></td>
<td>logspace, 56</td>
</tr>
<tr>
<td></td>
<td>matrix, 27, 29, 36</td>
</tr>
<tr>
<td></td>
<td>meshgrid, 59–60, 67, 70, 75, 77, 82, 88, 202</td>
</tr>
<tr>
<td></td>
<td>multiply, 27</td>
</tr>
<tr>
<td></td>
<td>ones, 26</td>
</tr>
<tr>
<td></td>
<td>random.randn, 26</td>
</tr>
<tr>
<td></td>
<td>random.seed, 26</td>
</tr>
<tr>
<td></td>
<td>reshape, 81</td>
</tr>
<tr>
<td></td>
<td>roots, 436, 438</td>
</tr>
<tr>
<td></td>
<td>sqrt, 382, 438</td>
</tr>
<tr>
<td></td>
<td>sum, 97</td>
</tr>
<tr>
<td></td>
<td>trace, 28</td>
</tr>
<tr>
<td></td>
<td>zeros, 26, 37, 70, 81, 202, 207, 215, 222, 382, 383, 388, 389, 392, 443</td>
</tr>
<tr>
<td></td>
<td>numpy.linalg</td>
</tr>
</tbody>
</table>
### INDEX

**eig**, 36–38, 97  
**inv**, 28, 29, 467  
**solve**, 29, 467  
**numpy.matrix**  
transpose, 28  

**os**  
chdir, 91, 92  
getcwd, 91  
listdir, 91  
system, 91

**plot**  
45  

**plotly**  
offline.plot, 73  
figure_factory  
create_trisurf, 72, 73

**print**  
22, 25–27, 29, 31, 298, 308

**quit**  
16

**range**  
34–36, 298

**rgb**  
72, 73

**roots**  
437

**round**  
43

**scipy.integrate**  
cumtrapz, 384  
fixed_quad, 385–386  
fixed_quad, 383  
ode, 320  
odeint, 299–302, 304–308, 311, 314  
quad, 383, 385–386, 388–392  
quadature, 383, 385–386  
romb, 384–385  
romberg, 383, 385–386  
simps, 384–385  
solve_ivp, 320  
trapz, 384–385, 387

**scipy.linalg**  
lu, 96  
lu_factor, 96  
lu_solve, 96  
scipy.optimize  
bisect, 436–437, 443  
newton, 436–438, 440

**skimage.measure.marching_cubes_lewiner**, 72

**sorted**, 96

**str**, 17, 43, 208, 217

**sys**  
getsizeof, 91–92

**type**, 17, 22–25, 27

**unicodedata**  
lookup, 61

**vpython**  
box, 83  
rate, 84  
sphere, 83  
text, 83  
vector, 83, 84  
zip, 72

**PYTHON constants**  
False, 17, 35  
math  
e, 22  
pi (π), 22  
None, 95  
numpy  
pi (π), 52, 59, 60, 69, 75, 87, 88, 207, 215, 381, 382, 385, 388, 392, 439–441  
sys.path, 91  
True, 17, 35

**PYTHON control structures**  
if/elif/else, 35–36  
try/except, 95  
while, 35, 39, 81, 84, 95

**PYTHON current directory**, see current directory in PYTHON

**PYTHON data attributes**, see also PYTHON attributes, 30

**PYTHON data types**, see also PYTHON commands, PYTHON scripts, 43–47

**PYTHON default directory**, see current directory in PYTHON

**PYTHON Edit Window**, 38–40

**PYTHON environment variables**  
MATPLOTLIBRC, 93  
PYTHONPATH, 91

**PYTHONSTART**, 92–93

**PYTHON functions**, see also PYTHON commands, PYTHON scripts, 43–47

**cos**, 391  
float, 40  
math  
acos, 21  
asin, 21  
atan, 21  
atan2, 21  
cos, 21  
acosh, 21  
erf, 21  
exp, 21
<table>
<thead>
<tr>
<th>Function/Module</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fabs</code></td>
<td>21</td>
</tr>
<tr>
<td><code>factorial</code></td>
<td>21</td>
</tr>
<tr>
<td><code>gamma</code></td>
<td>21</td>
</tr>
<tr>
<td><code>radians</code></td>
<td>21</td>
</tr>
<tr>
<td><code>asin</code></td>
<td>21</td>
</tr>
<tr>
<td><code>sqrt</code></td>
<td>21, 22</td>
</tr>
<tr>
<td><code>tan</code></td>
<td>21</td>
</tr>
<tr>
<td><code>atanh</code></td>
<td>21</td>
</tr>
<tr>
<td><code>sqrt</code></td>
<td>44, 56, 69, 308, 381–383, 385, 443</td>
</tr>
<tr>
<td><code>pow</code> (exponentiation)</td>
<td>20</td>
</tr>
<tr>
<td><code>quad</code></td>
<td>44</td>
</tr>
<tr>
<td><code>rings</code></td>
<td>391</td>
</tr>
<tr>
<td><code>round</code></td>
<td>20</td>
</tr>
<tr>
<td><code>sqrt</code></td>
<td>391</td>
</tr>
<tr>
<td><strong>user-defined</strong></td>
<td>43–47</td>
</tr>
<tr>
<td><strong>PYTHON instance attributes, see also</strong></td>
<td></td>
</tr>
<tr>
<td><strong>PYTHON attributes</strong></td>
<td>30</td>
</tr>
<tr>
<td><strong>PYTHON keywords, 18–19, 32, 33</strong></td>
<td></td>
</tr>
<tr>
<td><code>align</code></td>
<td>83</td>
</tr>
<tr>
<td><code>args</code></td>
<td>304, 305, 308, 311, 314, 390–392</td>
</tr>
<tr>
<td><code>atol</code></td>
<td>318</td>
</tr>
<tr>
<td><code>axis</code></td>
<td>37, 81, 444</td>
</tr>
<tr>
<td><code>azim</code></td>
<td>62</td>
</tr>
<tr>
<td><code>both</code></td>
<td>444</td>
</tr>
<tr>
<td><code>bottom</code></td>
<td>444</td>
</tr>
<tr>
<td><code>cmap</code></td>
<td>65, 70</td>
</tr>
<tr>
<td><code>colormap</code></td>
<td>72, 73</td>
</tr>
<tr>
<td><code>colors</code></td>
<td>63–64, 76, 89</td>
</tr>
<tr>
<td><code>cstride</code></td>
<td>61</td>
</tr>
<tr>
<td><code>divmax</code></td>
<td>393</td>
</tr>
<tr>
<td><code>ecolor</code></td>
<td>56</td>
</tr>
<tr>
<td><code>edgecolor</code></td>
<td>61, 62, 65, 68, 70, 82, 88</td>
</tr>
<tr>
<td><code>elev</code></td>
<td>62</td>
</tr>
<tr>
<td><code>epsabs</code></td>
<td>393</td>
</tr>
<tr>
<td><code>epsrel</code></td>
<td>393</td>
</tr>
<tr>
<td><code>fargs</code></td>
<td>80</td>
</tr>
<tr>
<td><code>filename</code></td>
<td>73</td>
</tr>
<tr>
<td><code>font</code></td>
<td>83</td>
</tr>
<tr>
<td><code>frames</code></td>
<td>79, 80, 82</td>
</tr>
<tr>
<td><code>height</code></td>
<td>83</td>
</tr>
<tr>
<td><code>hspace</code></td>
<td>54, 306</td>
</tr>
<tr>
<td><code>init_func</code></td>
<td>79, 80</td>
</tr>
<tr>
<td><code>interval</code></td>
<td>79, 80</td>
</tr>
<tr>
<td><code>labelbottom</code></td>
<td>444</td>
</tr>
<tr>
<td><code>labelsizetrue</code></td>
<td>303, 311, 315, 317, 388, 390, 438, 439, 441, 444</td>
</tr>
<tr>
<td><code>level</code></td>
<td>72</td>
</tr>
<tr>
<td><code>limit</code></td>
<td>393</td>
</tr>
<tr>
<td><code>linestyle</code></td>
<td>49, 51, 54, 56, 311</td>
</tr>
<tr>
<td><code>make_trail</code></td>
<td>83</td>
</tr>
<tr>
<td><code>marker</code></td>
<td>49</td>
</tr>
<tr>
<td><code>markeredgecolor</code></td>
<td>54, 57</td>
</tr>
<tr>
<td><code>markerfacecolor</code></td>
<td>54, 57</td>
</tr>
<tr>
<td><code>markersize</code></td>
<td>49</td>
</tr>
<tr>
<td><code>maxiter</code></td>
<td>386, 393</td>
</tr>
<tr>
<td><code>n</code></td>
<td>393</td>
</tr>
<tr>
<td><code>normalize</code></td>
<td>77</td>
</tr>
<tr>
<td><code>pos</code></td>
<td>83</td>
</tr>
<tr>
<td><code>projection</code></td>
<td>60, 62, 65, 70, 88</td>
</tr>
<tr>
<td><code>radius</code></td>
<td>83</td>
</tr>
<tr>
<td><code>repeat</code></td>
<td>79, 80, 82</td>
</tr>
<tr>
<td><code>rotation</code></td>
<td>89</td>
</tr>
<tr>
<td><code>rstride</code></td>
<td>61</td>
</tr>
<tr>
<td><code>rtol</code></td>
<td>318, 393</td>
</tr>
<tr>
<td><code>shade</code></td>
<td>61, 62, 70</td>
</tr>
<tr>
<td><code>show.colorbar</code></td>
<td>72, 73</td>
</tr>
<tr>
<td><code>size</code></td>
<td>83</td>
</tr>
<tr>
<td><code>title</code></td>
<td>72, 73</td>
</tr>
<tr>
<td><code>tol</code></td>
<td>393</td>
</tr>
<tr>
<td><code>top</code></td>
<td>444</td>
</tr>
<tr>
<td><code>trail</code></td>
<td>83</td>
</tr>
<tr>
<td><code>trail_color</code></td>
<td>54, 306</td>
</tr>
<tr>
<td><strong>PYTHON linestyle codes, 52</strong></td>
<td></td>
</tr>
<tr>
<td><strong>PYTHON marker codes, 55</strong></td>
<td></td>
</tr>
<tr>
<td><strong>PYTHON method attributes, see also PYTHON attributes, 30</strong></td>
<td></td>
</tr>
</tbody>
</table>
scipy.optimize, 435
skimage, 72
sys, 20, 91
unicodedata, 61, 65
vpython, 83
PYTHON programs, see PYTHON commands,
PYTHON functions, PYTHON scripts
PYTHON py-files, see PYTHON scripts,
PYTHON functions
PYTHON reserved words, 17–18
None, 24
PYTHON scripts
apollo.py, 319
dampharm.py, 319
damposc.py, 305, 319
decay.py, 302–303, 319
decaymod.py, 303–304, 319
drumhead.py, 82, 108–109
drvnosc.py, 319
drvnpend.py, 319
ExtractFileName.py, 531
gausint.py, 385
henon.py, 319
irrad.py, 206–208, 233
laplace.py, 202–203
largamp.py, 319
lineshape.py, 45, 46
lineshape1.py, 46
lorenz.py, 319
luplot, 44–45, 47
luplot.py, 45
moment.py, 386
onedshm.py, 319
pdens.py, 306–310, 320
plotradio.py, 41, 106
proj.py, 439, 440
projectile.py, 83–84, 110
qmlower.py, 443
qmshn.py, 313–317, 320
qmupper.py, 442
quadpole.py, 219, 249–250
rcfilter.py, 55, 106–107
read_irrad.py, 252–253
rossler.py, 320
spiralc.py, 388
spirals.py, 389
stdwaves.py, 310–313, 320
stwave.py, 107–108
turnpts.py, 437
turnptsp.py, 437
twodshm.py, 320
twooscil.py, 320
vandpol.py, 320
wavegd.py, 214–216, 244
PYTHON search path, 91
PYTHON special symbols
’ ’ (enclose strings), 18, 22
” ” (enclose strings), 18
’ ’ ’ ’ (comments), 16, 47
* (multiplication), 20, 21
** (exponentiation), 20, 21, 50, 59, 60, 69,
103, 308
+ (addition), 20, 21
- (subtraction), 20, 21
/ (division), 20, 21
: (range), 26, 41, 56, 70, 308, 444
; (combine statements), 16, 54
= (assignment), 18
== (equal to), 95
# (comments), 16, 47
% (remainder), 20
\ (continue line), 16, 50, 52
f (in format statements), 201
for expressing conditions, 35
j (complex number), 28–29
\n (end of line), 41–43, 91, 202, 208
nan (not a number), 28
PYTHON symbol codes, 55
PYTHON system variables, see PYTHON
constants
PYTHON variables
vpython.scene
autoscale, 83
background, 83
height, 83
title, 83
width, 83
Q
Q of resonant circuit, 101
quadratic formula, 44, 128, 426, 427, 470, 471
quadtrature, 372–380
adaptive algorithms, 377
assessing error in, 375–377, 380–383,
396–398, 402–404
C programs, 402–407
FORTRAN programs, 396–402
Gauss-Kronrod, 394
Gauss-Legendre, 380
Gaussian, 377–380, 382–383, 410, 413
iterative algorithms, 377
midpoint rule, 168, 372
Newton-Cotes, 372–377
recursive trapezoidal rule, 374
Romberg integration, 375
roundoff error, see roundoff error
Simpson’s rule, 168, 373, 375, 379, 382, 410,
413
trapezoidal rule, 372, 381, 396, 402
trunrection error, see truncation error
with MATHEMATICA, 393–396
with PYTHON, 380–392
quality factor, see Q of resonant circuit
quantum energy levels, see energy levels
quantum probabilities, 358
in hydrogen atom, 69, 102, 208–210, 228, 408, 411
of harmonic oscillator, 358, 387, 395, 411
quantum scattering, 163
quantum wave function, see wave function

R

radius of gyration
  definition of, 353
  of cardioid, 409
  of semicircular plate, 362
railroad problem, 473
range of projectiles, see trajectories
Rayleigh criterion, 474
reaction kinetics, 342
reading data
  from files
    in C, 221
    in FORTRAN, 220
    in pseudocode, 187
    in PYTHON, 40–41, 222
  from keyboard
    in C, 199–200
    in FORTRAN, 192–193
    in pseudocode, 174
    in PYTHON, 94–95
recasting
  algorithms, 224
  control structures, 223
recursion relationship, 285
redirection of output in UNIX, 191, 198
reflection at dielectric film, 99
relativity
  mass/momentum/energy, 99
  motion under constant force, 360–361
relaxation, 183–184
residual, 435
resonance
  in LRC circuit, 101, 268, 338
restoring initial state
  in MATHEMATICA, 121
  in PYTHON, 94
returning to command level
  in MATHEMATICA, 112
  in PYTHON, 16
returning to operating system
  from MATHEMATICA, 113
  from PYTHON, 16
ReVTeX, 519
Richardson extrapolation, 374, 381
Rodrigues’ formula, 162
Romberg integration, see quadrature
root finding
  accuracy in, 434–435
  by bisection, 432, 436–437, 452–455, 458–461
  by Laguerre’s method, 434
  by Müller’s method, 434
  by secant method, 434, 436–437, 446
  by trial and error, 450–452, 457–458, 468
  in C, 460–463, 469
  in FORTRAN, 453–457, 469
  in MATHEMATICA, 427–432, 444–450, 469
  in PYTHON, 435–444, 467, 469
roundoff error
  in numerical quadrature, 375
  in ODEs, 294
rubber lengths in ßÎ±, 489
Ruchardt’s experiment, 478
Runge-Kutta methods, 292–294
running program, see Local Guide

S

saving files, see writing data to files
scalar fields
  three-dimensional, 208–210
  in C, 209–210
  in FORTRAN, 209
  in PYTHON, 210
  two-dimensional, 203–208
  in C, 204–206
  in FORTRAN, 203–204
  in PYTHON, 206–208
search path
  in MATHEMATICA, 157
  in PYTHON, 91
secant method, see root finding
separating statements on one line
  in C, 196
  in MATHEMATICA, 112
  in pseudocode, 176
  in PYTHON, 16, 34, 54
series solutions to ODEs, 285–287
sets
  solutions to ODEs, 285–287
shooting method, 310
sigmoid curve, 266, 279
simple pendulum, 341, 354, 368–369, 412
Simpson’s rule, see quadrature
simultaneous equations
  by direct methods, 466
  by iterative methods, 466
  linear, 464–467
  nonlinear, 467–470
singular equations, 464
singular value decomposition, 466
solving Laplace’s equation, see Laplace’s
equation
solving linear equations, 464–467
INDEX

in MATHEMATICA, 129
in PYTHON, 29
solving ODEs
  with C, 329–336
  with FORTRAN, 322–329
  with MATHEMATICA, 274–287, 320–322
  with PYTHON, 297–320
source code, 172
special symbols
  in C, see C special symbols
  in FORTRAN, see FORTRAN special symbols
  in MATHEMATICA, see MATHEMATICA special symbols
  in PYTHON, see PYTHON special symbols
spherical coordinates
  in MATHEMATICA, 170
spurious roots, 419, 425, 443, 449
stable equilibrium, 420
standing waves
  in cube, 102, 228
Stark effect, 98, 166, 471
startup files, see initialization files
structure of data files, 184–185
subroutines
  in C, see C functions
  in FORTRAN, see FORTRAN subroutines
  in MATHEMATICA, see MATHEMATICA functions
  in PYTHON, see PYTHON functions, PYTHON scripts
suppressing evaluation
  in MATHEMATICA, 133–134
suppressing output
  in MATHEMATICA, 112
  in PYTHON, 22

T

tables
  in \LaTeX, 498–500
Taylor series
  for solving ODEs, 292–295
  for solving simultaneous equations, 469
termination of statements
  in C, 196
  in MATHEMATICA, 112
  in pseudocode, 176
  in PYTHON, 16
\LaTeX, 483, 491
\LaTeX contact information, 483, 535
\LaTeX output
  from MATHEMATICA, see \LaTeX output
texdoc, 503, 525
\TeXlive, 1
\TeXlive contact information, 535
text editors, see Local Guide
text in graphics, see placing text in graphics
TGIF, 1
TGIF contact information, 535
thermal equilibrium, 101
thin lens equation, 99
tikz, 503–506
tikz commands
  \definecolor, 504
  \draw, 504–505
  \foreach, 504, 505
  \node, 504, 505
  \x, 504, 505
trajectories, 351
  double pendulum, 337
  earth falling into sun, 408, 411
  exponentially decaying force, 407
  in central forces, 269–271, 306–310, 341
  in constant magnetic field, 85, 158, 228
  in Coulomb field, 338
  in crossed E and B, 341
  simple harmonic oscillator, 407
  under constant force, 360–361, 407
transformation rules
  in MATHEMATICA, 124, 128, 274, 284
transmission at dielectric film, 99
transpose
  in MATHEMATICA, 130, 145
  in PYTHON, 28
trapezoidal rule, see quadrature
tree-structured directories, 8
tridiagonal systems, 466, 477
truncation error
  global, 295
  in midpoint formula, 376
  in numerical quadrature, 375, 400, 405
  in ODEs, 294, 298–299
  in Simpson’s rule, 376
  in trapezoidal rule, 376, 381, 398, 403
  per step, 295
truth table, 175
tuples
  in PYTHON, 22–24
two’s complement notation, 6
two-way selection
  in C, 199, 459
  in FORTRAN, 192, 453
  in MATHEMATICA, 143–144
  in pseudocode, 176
  in PYTHON, 35–36

U

UNIX, 1
UNIX commands
aspell, 515–516
cc, 197, 258, 259, 330, 333, 335, 457, 459, 461, 462
cleanbak, 524
cleanTEX, 524
cp, 257, 258
dvipdfm, 488
dvips, 487
f77, 191, 257, 323, 327, 328, 398, 400, 451, 453, 454, 456
f90, 191
f95, 191
ispell, 515–516
latex, 486, 487
lp, 488
ls, 511
pdfcrop, 510–511
pdflatex, 487
pdfops, 511
ps2pdf, 488
python, 16
setenv, 92
texdoc, 503
xdvi, 488
UNIX shell scripts
ceps2pdf, 510–511, 530
cleanbak, 524
cleanTEX, 524
cps2pdf, 510–511, 530
dfileeps, 511, 530
dfileps, 511, 530
UNIX special symbols
> (output redirection), 191, 198
UNIX tutorial, see Local Guide
unstable equilibrium, 420
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 MATHEMATICA, see MATHEMATICA functions
in PYTHON, see PYTHON functions
user-defined subroutines
in C, see C functions
in FORTRAN, see FORTRAN subroutines

V

Van der Pol oscillator, 340
variable names
in C, 196
in FORTRAN, 187, 189
in MATHEMATICA, 114
in pseudocode, 172
in PYTHON, 17–18
vector arrays
three-dimensional, 217–219
two-dimensional, 211–216
vector calculus
with MATHEMATICA, 139–141
vector fields
three-dimensional
in C, 218–219
in FORTRAN, 218
in PYTHON, 219
two-dimensional
in C, 213–214
in FORTRAN, 212–213
in PYTHON, 214–216
vector identities, 170
velocity field
of rotating object, 229
vibrating string, 79–80
vim, 1
viscous damping, 263, 266

W

wave function, 272–273, 313–318, 358, 409, 411
Wheatstone bridge, 163
Windows, 1
Windows batch files
ceps2pdf.bat, 510–511, 530
cleanbak.bat, 524
cleanTEX.bat, 524
cps2pdf.bat, 510–511, 530
dfileeps.bat, 511, 529
dfileps.bat, 511, 529
Windows commands
cleanbak, 524
cleanTEX, 524
dir, 511
dvipdfm, 488
dvips, 487
latex, 486, 487
pdflatex, 487
pdfcrop, 510–511
pdfops, 511
print, 488
ps2pdf, 488
python, 16
set, 92
setx, 92
texdoc, 503
yap, 488
Winedt, 1
Winedt contact information, 535
writing data to files, 228
writing data to screen
in C, 195, 197
<table>
<thead>
<tr>
<th>INDEX</th>
<th>557</th>
</tr>
</thead>
<tbody>
<tr>
<td>in FORTRAN, 188, 190</td>
<td>Xemacs, 1</td>
</tr>
<tr>
<td>in pseudocode, 174</td>
<td>Xemacs contact information, 536</td>
</tr>
<tr>
<td>in PYTHON, 20–24</td>
<td>Xv contact information, 536</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>X</th>
<th>yap, 488, 514–515</th>
</tr>
</thead>
<tbody>
<tr>
<td>xdvi, 488, 514–515</td>
<td></td>
</tr>
</tbody>
</table>